C++20

The Complete Guide

Nicolai M. Josuttis

To the nation of Ukraine
and all the children, women, men, and soldiers
killed in a terrible war driven by a Russian dictator

C++20 - The Complete Guide

First Edition

Nicolai M. Josuttis

This version was published on **2022-10-30**.

This book was typeset by Nicolai M. Josuttis using the LaTeX document processing system.

Contents

Preface

C++20 is the next evolution in modern C++ programming, and it is already (partially) supported by the latest version of GCC, Clang, and Visual C++. The move to C++20 is at least as big a step as the move to C++11. C++20 contains a significant number of new language features and libraries that will again change the way we program in C++. This applies to both application programmers and programmers who provide foundation libraries.

An Experiment

This book is an experiment in two ways:

- I am writing an in-depth book that covers complex new features invented and provided by different programmers and C++ working groups. However, I can ask questions and I do.
- I am publishing the book myself on Leanpub and for printing on demand. That is, this book is written step by step and I will publish new versions as soon there is a significant improvement that makes the publication of a new version worthwhile.

The good thing is:

- You get the view of the language features from an experienced application programmer—somebody who feels the pain a feature might cause and asks the relevant questions to be able to explain the motivation for a feature, its design, and all consequences for using it in practice.
- You can benefit from my experience with C++20 while I am still learning and writing.
- This book and all readers can benefit from your early feedback.

This means that you are also part of the experiment. So help me out: give feedback about flaws, errors, features that are not explained well, or gaps, so that we all can benefit from these improvements.

Acknowledgments

This book would not have been possible without the help and support of a huge number of people.

First of all, I would like to thank you, the C++ community, for making this book possible. The incredible design of all the features of C++20, the helpful feedback, and your curiosity are the basis for the evolution of a successful language. In particular, thanks for all the issues you told me about and explained and for the feedback you gave.

I would especially like to thank everyone who reviewed drafts of this book or corresponding slides and provided valuable feedback and clarification. These reviews increased the quality of the book significantly, again proving that good things are usually the result of collaboration between many people. Therefore, so far (this list is still growing) huge thanks to Carlos Buchart, Javier Estrada, Howard Hinnant, Yung-Hsiang Huang, Daniel Krügler, Dietmar Kühl, Jens Maurer, Paul Ranson, Thomas Symalla, Steve Vinoski, Ville Voutilainen. Andreas Weis, Hui Xie, Leor Zolman, and Victor Zverovich.

In addition, I would like to thank everyone on the C++ standards committee. In addition to all the work involved in adding new language and library features, these experts spent many, many hours explaining and discussing their work with me, and they did so with patience and enthusiasm. Special thanks here go to Howard Hinnant, Tom Honermann, Tomasz Kaminski, Peter Sommerlad, Tim Song, Barry Revzin, Ville Voutilainen, and Jonathan Wakely.

Special thanks go to the LaTeX community for a great text system and to Frank Mittelbach for solving my LaTeX issues (it was almost always my fault).

And finally, many thanks go to my proofreader, Tracey Duffy, who has again done a tremendous job of converting my "German English" into native English.

Versions of This Book

Because this book is written incrementally, the following is a history of the major updates (newest first):

- **2022-10-29**: Describe missing details about floating-point and `struct` values as NTTPs.
- **2022-10-29**: Describe missing hours utilities.
- **2022-10-28**: Layout all pages.
- **2022-10-15**: Describe missing details of the join view.
- **2022-10-06**: Describe missing issues with standard views and list them all together.
- **2022-09-28**: Describe more range/iterator utility details (move sentinels, `iter_move()`, `iter_swap()`).
- **2022-09-22**: Update span documentation and document span operations in detail.
- **2022-09-16**: Fix the utilities for layout compatibility.
- **2022-08-22**: Add complete example programs and fix view details for each view.
- **2022-08-21**: Describe details of the istream view that were added to C++20 as a late fix.
- **2022-08-17**: Describe the new result type for range algorithms.
- **2022-08-05**: Describe how to deal with a non-existing timezone database.
- **2022-07-29**: Improve the description of `const` issues with views.
- **2022-07-27**: Describe the effect of caching views in detail.
- **2022-07-21**: Describe new iterator categories and `iterator_concept`.
- **2022-07-06**: Describe concepts for non-type template parameters.
- **2022-06-28**: Describe using `requires` for member functions to change APIs.
- **2022-06-19**: Describe the fix to the formatting library for compile-time parsing of format strings.
- **2022-05-31**: Split coroutine description into two chapters.
- **2022-05-24**: Describe properties and details of all views.
- **2022-05-23**: Describe coroutine exception handling.
- **2022-05-24**: Describe nested coroutines and symmetric transfer.
- **2022-05-19**: Describe asynchronous communication and thread pools for coroutines.

- **2022-05-07**: Describe jthread and stop token support for condition variables.
- **2022-04-28**: Describe owning views, which were added to C++20 as a late fix.
- **2022-04-24**: Describe significant details of coroutines and their awaiters.
- **2022-04-21**: Describe functions for safe integral comparisons.
- **2022-04-21**: Describe `std::ranges::swap()`.
- **2022-04-20**: Describe the new shift algorithms and the new min/max algorithms for ranges.
- **2022-04-20**: Describe `compare_three_way` and `lexicographical_compare_three_way()`.
- **2022-04-19**: Describe feature test macros.
- **2022-04-18**: Describe several additional details and clarifications about modules.
- **2022-04-09**: Describe deferred evaluation with `auto` parameters.
- **2022-03-26**: Describe awaiters, memory management, and other details of coroutines.
- **2022-03-08**: Describe details of utilities of the ranges library.
- **2022-03-03**: Rewrite the coroutines chapter with several more details.
- **2022-02-20**: Describe the range-based `for` loop with initialization.
- **2022-02-17**: Reorganize book chapters.
- **2022-02-16**: Describe new attributes and attribute extensions.
- **2022-02-15**: Describe conditional `explicit`.
- **2022-02-15**: Fix the way user-defined formatters should be implemented.
- **2022-02-12**: Describe `using` for enumeration values.
- **2022-02-09**: Add figure with subset of the subsumptions of the standard concepts.
- **2022-01-08**: Describe class template argument deduction for aggregates.
- **2022-01-05**: Describe modified definition of aggregates.
- **2022-01-04**: Describe aggregate initialization with parentheses.
- **2021-12-31**: Describe the new iterator traits (such as `std::iter_value_t`).
- **2021-12-31**: Describe where `typename` is no longer required for type members of template parameters.
- **2021-12-30**: Describe new bit operations (including `bit_cast<>()`).
- **2021-12-29**: Describe improvements for string types (including `std::u8string` and using strings at compile time).
- **2021-12-28**: Describe the `unseq` execution policy for algorithms.
- **2021-12-25**: Describe all other missing lambda features.
- **2021-12-11**: Describe `consteval` lambdas.
- **2021-12-06**: Describe `std::endian`.
- **2021-12-06**: Describe synchronized output streams.
- **2021-12-04**: Describe the header file `<version>`.
- **2021-11-21**: Describe compile-time use of vectors and `constexpr` extensions.
- **2021-10-25**: Describe designated initializers.
- **2021-10-14**: Describe severe compatibility issues with `operator==`.
- **2021-10-14**: Describe mathematical constants.
- **2021-10-12**: Describe `constinit`, `consteval`, and `std::is_constant_evaluated()`.
- **2021-10-07**: Describe `char8_t` for UTF-8 (and its compatibility issues).
- **2021-10-02**: Clarify which `const` views you cannot iterate over.
- **2021-10-01**: Provide details about the formatting library.
- **2021-09-21**: Add updates and fixes according to several reviews.
- **2021-09-20**: Provide a first description of features for modules.

- **2021-09-11**: Discuss `const` issues of ranges and views.
- **2021-08-30**: All concepts are documented.
- **2021-08-28**: Document all new type traits.
- **2021-08-27**: Document iterator utility functions for ranges.
- **2021-08-26**: Document `lazy_split_view<>`, which was added to C++20 as a late fix.
- **2021-08-25**: Document all missing views of C++20 and `all_t<>`.
- **2021-08-20**: Document all new iterator and sentinel types.
- **2021-08-20**: Document all primary range adaptors (`counted()`, `all()`, and `common()`).
- **2021-08-19**: Document `common_view`.
- **2021-08-16**: Document how to deal with semantic constraints of concepts.
- **2021-08-15**: Document `type_identity`.
- **2021-07-31**: Document `empty_view`, `ref_view`, and `view_interface`.
- **2021-07-29**: Generic code for ranges has to use universal/forwarding references.
- **2021-07-28**: Document `iota_view` and `unreached_sentinel`.
- **2021-07-27**: Extend and reorganize chapters about ranges and views.
- **2021-07-17**: Document `subrange`.
- **2021-07-09**: Document `std::source_location`.
- **2021-06-29**: The initial published version of the book. The following features are more or less completely described or provide at least a full conceptual introduction:
  - Comparisons and the spaceship operator `<=>`
  - Generic functions
  - Concepts and requirements (details open)
  - Ranges and views (details open)
  - Spans
  - Non-type template parameter extensions
  - Formatted output (overview only)
  - Dates and timezones for `<chrono>`
  - Coroutines (first examples)
  - `std::thread` and stop tokens
  - New concurrency features

About This Book

This book presents all the new language and library features of C++20. It covers the motivation and context of each new feature with examples and background information.

As usual for my books, the focus lies on the application of the new features in practice and the book demonstrates how features impact day-to-day programming and how you can benefit from them in projects. This applies to both application programmers and programmers who provide generic frameworks and foundation libraries.

What You Should Know Before Reading This Book

To get the most from this book, you should already be familiar with C++. You should be familiar with the concepts of classes and references in general, and you should be able to write C++ code using components of the C++ standard library, such as IOStreams and containers. You should also be familiar with the basic features of *Modern C++*, such as `auto` or the range-based `for` loop and other basic features introduced with C++11, C++14, and C++17.

However, you do not have to be an expert. My goal is to make the content understandable for the average C++ programmer who does not necessarily know all the details or all of the latest features. I discuss basic features and review more subtle issues as the need arises.

This ensures that the text is accessible to experts and intermediate programmers alike.

Overall Structure of the Book

This book covers *all* changes to C++ introduced with C++20. This applies to both language and library features as well as both features that affect day-to-day application programming and features for the sophisticated implementation of (foundation) libraries. However, the more general cases and examples usually come first.

The different chapters are grouped, but the grouping has no deeper didactic reasoning other than that it makes sense to first introduce features that might be used by the subsequent features introduced afterwards. In principle, you can read the chapters in any order. If features from different chapters are combined, there are corresponding cross-references.

How to Read This Book

Do not be intimidated by the number of pages in this book. As always with C++, things can become pretty complicated when you look into details. For a basic understanding of the new features, the introductory motivations and examples are often sufficient.

In my experience, the best way to learn something new is to look at examples. Therefore, you will find a lot of examples throughout the book. Some are just a few lines of code that illustrate an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples are introduced by a C++ comment that describes the file that contains the program code. You can find these files on the website for this book at `http://www.cppstd20.com`.

The Way I Implement

Note the following hints about the way I write code and comments in this book.

Initializations

I usually use the modern form of initialization (introduced in C++11 as *uniform initialization*) with curly braces or with = in trivial cases:

```
int i = 42;
std::string s{"hello"};
```

The *brace initialization* has the following advantages:

- It can be used with fundamental types, class types, aggregates, enumeration types, and `auto`
- It can be used to initialize containers with multiple values
- It can detect narrowing errors (e.g., initialization of an `int` by a floating-point value)
- It cannot be confused with function declarations or calls

If the braces are empty, the default constructors of (sub)objects are called and fundamental data types are guaranteed to be initialized with `0`/`false`/`nullptr`.

Error Terminology

I often talk about programming errors. If there is no special hint, the term *error* or a comment such as

```
...      // ERROR
```

means a compile-time error. The corresponding code should not compile (with a conforming compiler).

If I use the term *runtime error*, the program might compile but not behave correctly or result in undefined behavior (thus, it might or might not do what is expected).

Code Simplifications

I try to explain all features with helpful examples. However, to concentrate on the key aspects to be taught, I might often skip other details that should be part of the code.

- Most of the time I use an ellipsis ("...") to signal additional code that is missing. Note that I do not use code font here. If you see an ellipsis with code font, code must have these three dots as a language feature (such as for "`typename...`").
- In header files, I usually skip the preprocessor guards. All header files should have something like the following:

  ```
  #ifndef MYFILE_HPP
  #define MYFILE_HPP
  ...
  #endif  //MYFILE_HPP
  ```

  Therefore, please beware and fix the code when using these header files in your projects.

The C++ Standards

C++ has different versions defined by different C++ standards.

The original C++ standard was published in 1998 and was subsequently amended by a *technical corrigendum* in 2003, which provided minor corrections and clarifications to the original standard. This "old C++ standard" is known as C++98 or C++03.

The world of "Modern C++" began with C++11 and was extended with C++14 and C++17. The international C++ standards committee now aims to issue a new standard every three years. Clearly, that leaves less time for massive additions, but it brings the changes to the broader programming community more quickly. The development of larger features, therefore, takes time and might cover multiple standards.

C++20 is now the beginning of the next "Even more Modern C++" evolution. Again, several ways of programming will probably change. However, as usual, compilers need some time to provide the latest language features. At the time of writing this book, C++20 is already at least partially supported by major compilers. However, as usual, compilers differ greatly in their support of new different language features. Some will compile most or even all of the code in this book, while others may only be able to handle a significant subset. I expect that this problem will soon be resolved as programmers everywhere demand standard support from their vendors.

Example Code and Additional Information

You can access all example programs and find more information about this book from its website, which has the following URL:

`http://www.cppstd20.com`

Feedback

I welcome your constructive input—both negative and positive. I have worked very hard to bring you what I hope you will find to be an excellent book. However, at some point I had to stop writing, reviewing, and tweaking to "release the new revision." You may therefore find errors, inconsistencies, presentations that could be improved, or topics that are missing altogether. Your feedback gives me a chance to fix these issues, inform all readers about the changes through the book's website, and improve any subsequent revisions or editions.

The best way to reach me is by email. You will find the email address on the website for this book:

`http://www.cppstd20.com`

If you use the ebook, you might want to ensure you to have the latest version of this book available (remember it is written and published incrementally). You should also check the book's website for the currently known errata before submitting reports. Please always refer to the publishing date of this version when giving feedback. The current publishing date is **2022-10-30** (you can also find it on page ii, the page directly after the cover).

Many thanks.

Chapter 1

Comparisons and Operator <=>

C++20 simplifies the definition of comparisons for user-defined types and introduces better ways of dealing with them. The new operator <=> (also called the *spaceship* operator) was introduced for this purpose.

This chapter explains how to define and deal with comparisons since C++20 using these new features.

1.1 Motivation for Operator<=>

Let us look first at the motivation for the new way comparisons are handled since C++20 and the new operator <=>.

1.1.1 Defining Comparison Operators Before C++20

Before C++20, you had to define six operators for a type to provide full support for all possible comparisons of its objects.

For example, if you wanted to compare objects of a type `Value` (having an integral ID), you had to implement the following:

```
class Value {
 private:
  long id;
  ...
 public:
  ...
  // equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id;        // basic check for equality
  }
  bool operator!= (const Value& rhs) const {
    return !(*this == rhs);     // derived check
  }
```

```
    // relational operators:
    bool operator< (const Value& rhs)  const {
      return id < rhs.id;         // basic check for ordering
    }
    bool operator<= (const Value& rhs) const {
      return !(rhs < *this);      // derived check
    }
    bool operator> (const Value& rhs)  const {
      return rhs < *this;         // derived check
    }
    bool operator>= (const Value& rhs) const {
      return !(*this < rhs);      // derived check
    }
  };
```

This enables you to call any of the six comparison operators for a `Value` (the object the operator is defined for) with another `Value` (passed as parameter `rhs`). For example:

```
  Value v1, v2;
  ... ;
  if (v1 <= v2) {     // calls v1.operator<=(v2)
    ...
  }
```

The operators might also be called indirectly (e.g., by calling `sort()`):

```
  std::vector<Value> coll;
  ... ;
  std::sort(coll.begin(), coll.end());  // uses operator < to sort
```

Since C++20 , the call might alternatively use ranges:

```
  std::ranges::sort(coll);              // uses operator < to sort
```

The problem is that even though most of the operators are defined in terms of other operators (they are all based on either operator == or operator <), the definitions are tedious and they add a lot of visual clutter.

In addition, for a well-implemented type, you might need more:

- Declare the operators with `noexcept` if they cannot throw
- Declare the operators with `constexpr` if they can be used at compile time
- Declare the operators as "hidden friends" (declare them with `friend` inside the class structure so that both operands become parameters and support implicit type conversions) if the constructors are not `explicit`
- Declare the operators with `[[nodiscard]]` to force warnings if the return value is not used

For example:

lang/valueold.hpp

```
class Value {
 private:
  long id;
  ...
 public:
  constexpr Value(long i) noexcept   // supports implicit type conversion
   : id{i} {
  }
  ...
  // equality operators:
  [[nodiscard]] friend constexpr
  bool operator== (const Value& lhs, const Value& rhs) noexcept {
    return lhs.id == rhs.id;        // basic check for equality
  }
  [[nodiscard]] friend constexpr
  bool operator!= (const Value& lhs, const Value& rhs) noexcept {
    return !(lhs == rhs);           // derived check for inequality
  }

  // relational operators:
  [[nodiscard]] friend constexpr
  bool operator< (const Value& lhs, const Value& rhs) noexcept {
    return lhs.id < rhs.id;         // basic check for ordering
  }
  [[nodiscard]] friend constexpr
  bool operator<= (const Value& lhs, const Value& rhs) noexcept {
    return !(rhs < lhs);            // derived check
  }
  [[nodiscard]] friend constexpr
  bool operator> (const Value& lhs, const Value& rhs) noexcept {
    return rhs < lhs;               // derived check
  }
  [[nodiscard]] friend constexpr
  bool operator>= (const Value& lhs, const Value& rhs) noexcept {
    return !(lhs < rhs);            // derived check
  }
};
```

1.1.2 Defining Comparison Operators Since C++20

Since C++20, a couple of things have changed regarding comparison operators.

Operator == Implies Operator !=

To check for inequality, it is now enough to define operator ==.

When the compiler finds no matching declaration for an expression `a!=b`, the compiler *rewrites* the expressions and looks for `!(a==b)` instead. If that does not work, the compiler also tries to change the order of the operands, so it also tries `!(b==a)`:

```
a != b   // tries: a!=b, !(a==b), and !(b==a)
```

Therefore, for `a` of `TypeA` and `b` of `TypeB`, the compiler will be able to compile

```
a != b
```

It can do this if there is

- A free-standing `operator!=(TypeA, TypeB)`
- A free-standing `operator==(TypeA, TypeB)`
- A free-standing `operator==(TypeB, TypeA)`
- A member function `TypeA::operator!=(TypeB)`
- A member function `TypeA::operator==(TypeB)`
- A member function `TypeB::operator==(TypeA)`

Directly calling a defined operator `!=` is preferred (but the order of the types has to fit). Changing the order of the operands has the lowest priority. Having both a free-standing and a member function is an ambiguity error.

Thus, with

```
bool operator==(const TypeA&, const TypeB&);
```

or

```
class TypeA {
 public:
  ...
  bool operator==(const TypeB&) const;
};
```

the compiler will be able to compile:

```
MyType a;
MyType b;
...
a == b;   // OK: fits perfectly
b == a;   // OK, rewritten as: a == b
a != b;   // OK, rewritten as: !(a == b)
b != a;   // OK, rewritten as: !(a == b)
```

Note that thanks to rewriting, implicit type conversions for the first operand are also possible when rewriting converts the operand so that it becomes the parameter of the defined member function.

See the example `sentinel1.cpp` for how to benefit from this feature by only defining a member operator == when `!=` with a different order of operands is called.

Operator <=>

There is no equivalent rule that for all relational operators it is enough to have operator < defined. However, you only have to define the new operator <=>.

In fact, the following is enough to enable programmers to use all possible comparators:

lang/value20.hpp

```
#include <compare>

class Value {
 private:
  long id;
  ...
 public:
  constexpr Value(long i) noexcept
   : id{i} {
  }
  ...
  // enable use of all equality and relational operators:
  auto operator<=> (const Value& rhs) const  = default;
};
```

In general, operator == handles the equality of objects by defining == and `!=`, while operator <=> handles the order of objects by defining the relational operators. However, by declaring an `operator<=>` with `=default`, we use a special rule that a ***defaulted*** member `operator<=>`:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
};
```

generates a corresponding member `operator==`, so that we effectively get:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
  auto operator== (const Value& rhs) const  = default;  // implicitly generated
};
```

The effect is that both operators use their default implementation, which compares objects member by member. This means that the order of the members in the class matters.

Thus, with

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const  = default;
};
```

we get all we need to be able to use all six comparison operators.

In addition, even when declaring the operator as a member function, the following applies for the generated operators:

- They are `noexcept` if comparing the members never throws
- They are `constexpr` if comparing the members is possible at compile time
- Thanks to *rewriting*, implicit type conversions for the first operand are also supported

This reflects that, in general, `operator==` and `operator<=>` handle different but related things:

- `operator==` defines equality and can be used by the *equality operators* `==` and `!=`.
- `operator<=>` defines the ordering and can be used by the *relational operators* `<`, `<=`, `>`, and `>=`.

Note that you have to include header `<compare>` when defaulting or using operator `<=>`.

```
#include <compare>
```

However, most header files for standard types (strings, containers, `<utility>`) include this header anyway.

Implementing Operator <=>

To have more control over the generated comparison operators, you can define `operator==` and `operator<=>` yourself. For example:

lang/value20def.hpp

```
#include <compare>

class Value {
 private:
  long id;
  ...
 public:
  constexpr Value(long i) noexcept
   : id{i} {
  }
  ...
  // for equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id;         // defines equality (== and !=)
  }
  // for relational operators:
  auto operator<=> (const Value& rhs) const {
    return id <=> rhs.id;        // defines ordering (<, <=, >, and >=)
  }
};
```

This means that you can specify which members in which order matter or implement special behavior.

The way these basic operators work is that if an expression uses one of the comparison operators and does not find a matching direct definition, the expression is *rewritten* so that it can use these operators.

Corresponding to rewriting calls of equality operators, rewriting might also change the order of relational operands, which might enable implicit type conversion for the first operand. For example, if

```
x <= y
```

does not find a matching definition of `operator<=`, it might be rewritten as

```
(x <=> y) <= 0
```

or even

```
0 <= (y <=> x)
```

As you can see by this rewriting, the new `operator<=>` performs a three-way comparison, which yields a value you can compare with `0`:

- If the value of `x<=>y` is equal to `0`, `x` and `y` are equal or equivalent.
- If the value of `x<=>y` is less than `0`, `x` is less than `y`.
- If the value of `x<=>y` is greater than `0`, `x` is greater than `y`.

However, note that the return type of `operator<=>` is not an integral value. The return type is a type that signals the *comparison category*, which could be *strong ordering*, *weak ordering*, or *partial ordering*. These types support the comparison with `0` to deal with the result.

1.2 Defining and Using Comparisons

The following sections explain the details of the handling of the comparison operators since C++20.

1.2.1 Using Operator<=>

Operator `<=>` is a new binary operator. It is defined for all fundamental data types for which the relational operators are defined. As usual, it can be user-defined as `operator<=>()`.

Operator `<=>` takes precedence over all other comparison operators, which means that you need parentheses to use it in an output statement but not to compare its result with another value:

```
std::cout << (0 < x <=> y) << '\n';    // calls 0 < (x <=> y)
```

Please note that you have to include a specific header file to deal with the result of operator `<=>`:

```
#include <compare>
```

This applies to declaring it (as defaulted), implementing it, or using it. For example:

```
#include <compare>    // for calling <=>

auto x = 3 <=> 4;     // does not compile without header <compare>
```

Most header files for standard types (strings, containers, `<utility>`) include this header anyway. However, to call the operator on values or types that do not require this header, you have to include `<compare>`.

Note that operator `<=>` is for implementing types. Outside the implementation of an `operator<=>`, programmers should never invoke `<=>` directly. Although you can, you should never write `a<=>b < 0` instead of `a<b`.

1.2.2 Comparison Category Types

The new operator `<=>` does not return a Boolean value. Instead, it acts similarly to three-way-comparisons yielding a negative value to signal *less*, a positive value to signal *greater*, and `0` to signal *equal* or *equivalent*. This behavior is similar to the return value of the C function `strcmp()`; however, there is an important difference: the return value is not an integral value. Instead, the C++ standard library provides three possible return types, which reflect the category of the comparison.

Comparison Categories

When comparing two values to put them in an order, we have different categories of behavior that could happen:

- With **strong ordering** (also called *total ordering*), any value of a given type is *less* than or *equal* to or *greater* than any other value of this type (including itself).

  Typical examples of this category are integral values or common string types. A string `s1` is less than or equal to or greater than a string `s2`.

  If a value of this category is neither less than nor greater than another value, both values are equal. If you have multiple objects, you can sort them in ascending or descending order (with equal values having any order among each other).
- With **weak ordering**, any value of a given type is *less* than or ***equivalent*** to or *greater* than any other value of this type (including itself). However, equivalent values do not have to be equal (have the same value).

  A typical example of this category is a type for case-insensitive strings. A string `"hello"` is less than `"hello1"` and greater than `"hell"`. However, `"hello"` is equivalent to `"HELLO"` although these two strings are not equal.

  If a value of this category is neither less than nor greater than another value, both values are at least equivalent (they *might* even be equal). If you have multiple objects, you can sort them in ascending or descending order (with equivalent values having any order among each other).
- With **partial ordering**, any value of a given type *could* be *less* than or *equivalent* to or *greater* than any other value of this type (including itself). However, in addition, it may not be possible to specify a specific order between two values at all.

  A typical example of this category are floating-point types, because they might have the special value NaN ("not a number"). Any comparison with NaN yields `false`. Therefore, in this case a comparison might yield that two values are *unordered* and the comparison operator might return one of four values.

  If you have multiple objects, you might not be able to sort them in ascending or descending order (unless you ensure that values that cannot be ordered are not there).

Comparison Category Types in the Standard Library

For the different comparison categories, the C++20 standard introduces the following types:

- `std::`**`strong_ordering`** with the values:
  - `std::`**`strong_ordering::less`**
  - `std::`**`strong_ordering::equal`**
    (also available as `std::`**`strong_ordering::equivalent`**)
  - `std::`**`strong_ordering::greater`**

- `std::`**`weak_ordering`** with the values:
  - `std::`**`weak_ordering::less`**
  - `std::`**`weak_ordering::equivalent`**
  - `std::`**`weak_ordering::greater`**
- `std::`**`partial_ordering`** with the values:
  - `std::`**`partial_ordering::less`**
  - `std::`**`partial_ordering::equivalent`**
  - `std::`**`partial_ordering::greater`**
  - `std::`**`partial_ordering::unordered`**

Note that all types have the values `less`, `greater`, and `equivalent`. However, `strong_ordering` also has `equal`, which is the same as `equivalent` there, and `partial_ordering` has the value `unordered`, representing neither less nor equal nor greater.

Stronger comparison types have implicit type conversions to weaker comparison types. This means that you can use any `strong_ordering` value as `weak_ordering` or `partial_ordering` value (`equal` then becomes `equivalent`).

1.2.3 Using Comparison Categories with `operator<=>`

The new operator <=> should return a value of one of the comparison category types, representing the result of the comparison combined with the information about whether this result is able to create a strong/total, weak, or partial ordering.

For example, this is how a free-standing `operator<=>` might be defined for a type `MyType`:

```
std::strong_ordering operator<=> (MyType x, MyOtherType y)
{
  if (xIsEqualToY)  return std::strong_ordering::equal;
  if (xIsLessThanY)  return std::strong_ordering::less;
  return std::strong_ordering::greater;
}
```

Or, as a more concrete example, defining `operator<=>` for a type `MyType`:

```
class MyType {
  ...
  std::strong_ordering operator<=> (const MyType& rhs) const {
    return value == rhs.value ? std::strong_ordering::equal :
           value < rhs.value ? std::strong_ordering::less :
                               std::strong_ordering::greater;
  }
};
```

However, it is usually easier to define the operator by mapping it to results of underlying types. Therefore, it would be better for the member `operator<=>` above to just yield the value and category of its member `value`:

```
class MyType {
  ...
  auto operator<=> (const MyType& rhs) const {
    return value <=> rhs.value;
  }
};
```

This not only returns the right value; it also ensures that the return value has the right comparison category type depending on the type of the member `value`.

1.2.4 Calling Operator <=> Directly

You can call any defined operator <=> directly:

```
MyType x, y;
...
x <=> y                  // yields a value of the resulting comparison category type
```

As written, you should only call operator <=> directly when implementing `operator<=>`. However, it can be very helpful to know the returned comparison category.

As also written, `operator<=>` is predefined for all fundamental types for which the relational operators are defined. For example:

```
int x = 17, y = 42;
x <=> y                  // yields std::strong_ordering::less
x <=> 17.0               // yields std::partial_ordering::equivalent
&x <=> &x                // yields std::strong_ordering::equal
&x <=> nullptr           // ERROR: relational comparison with nullptr not supported
```

In addition, all types of the C++ standard library that provide relational operators also provide `operator<=>` now. For example:

```
std::string{"hi"} <=> "hi"                  // yields std::strong_ordering::equal;
std::pair{42, 0.0} <=> std::pair{42, 7.7}   // yields std::partial_ordering::less
```

For your own type(s), you only have to define `operator<=>` as a member or free-standing function.

Because the return type depends on the comparison category, you can check against a specific return value:

```
if (x <=> y == std::partial_ordering::equivalent)   // always OK
```

Due to the implicit type conversions to weaker ordering types, this will even compile if `operator<=>` yields a `strong_ordering` or `weak_ordering` value.

The other way around does not work. If the comparison yields a `weak_ordering` or `partial_ordering` value, you cannot compare it with a `strong_ordering` value.

```
if (x <=> y == std::strong_ordering::equal)   // might not compile
```

However, a comparison with `0` is always possible and usually easier:

```
if (x <=> y == 0)          // always OK
```

In addition, `operator<=>` might be called indirectly due to the new rewriting of relational operator calls:

```
if (!(x < y || y < x))     // might call operator<=> to check for equality
```

Or:

```
if (x <= y && y <= x)     // might call operator<=> to check for equality
```

Note that `operator!=` is never rewritten to call `operator<=>`. However, it might call an `operator==` member that is implicitly generated due to a defaulted `operator<=>` member.

1.2.5 Dealing with Multiple Ordering Criteria

To compute the result of `operator<=>` based on multiple attributes, you can usually implement just a chain of sub-comparisons until the result is not equal/equivalent or you reach the final attribute to be compared:

```
class Person {
  ...
  auto operator<=> (const Person& rhs) const {
    auto cmp1 = lastname <=> rhs.lastname;     // primary member for ordering
    if (cmp1 != 0) return cmp1;                // return result if not equal
    auto cmp2 = firstname <=> rhs.firstname;   // secondary member for ordering
    if (cmp2 != 0) return cmp2;                // return result if not equal
    return value <=> rhs.value;                // final member for ordering
  }
};
```

However, the return type does not compile if the attributes have different comparison categories. For example, if a member `name` is a string and a member `value` is a `double`, we have conflicting return types:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const {  // ERROR: different return types deduced
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // return strong_ordering for std::string
    return value <=> rhs.value;     // return partial_ordering for double
  }
};
```

In that case, you can use a conversion to the weakest comparison type. If you know the weakest comparison type, you can just declare it as the return type:

```
class Person {
  std::string name;
  double value;
  ...
  std::partial_ordering operator<=> (const Person& rhs) const {  // OK
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // strong_ordering converted to return type
    return value <=> rhs.value;     // partial_ordering used as the return type
  }
};
```

If you do not know the comparison types (e.g., their type is a template parameter), you can use a new type trait `std::common_comparison_category<>` that computes the strongest comparison category:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const    // OK
   -> std::common_comparison_category_t<decltype(name <=> rhs.name),
                                        decltype(value <=> rhs.value)> {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;     // used as or converted to common comparison type
    return value <=> rhs.value;     // used as or converted to common comparison type
  }
};
```

By using the trailing return type syntax (with `auto` in front and the return type after `->`), we can use the parameters to compute the comparison types. Even though in this case, you could just use `name` instead of `rhs.name`, this approach works in general (e.g., also for free-standing functions).

If you want to provide a stronger category than the one that is used internally, you have to map all possible values of the internal comparsions to values of the return type. This might include some error handling if you cannot map some values. For example:

```
class Person {
  std::string name;
  double value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;      // return strong_ordering for std::string
    auto cmp2 = value <=> rhs.value; // might be partial_ordering for double
    // map partial_ordering to strong_ordering:
    assert(cmp2 != std::partial_ordering::unordered);  // RUNTIME ERROR if unordered
    return cmp2 == 0 ? std::strong_ordering::equal
                     : cmp2 > 0 ? std::strong_ordering::greater
                                : std::strong_ordering::less;
  }
};
```

The C++ standard library provides some helper function objects for this. For example, to map floating-point values, you can call `std::strong_order()` for the two values to be compared:

```
class Person {
  std::string name;
  double value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;       // return strong_ordering for std::string
    // map floating-point comparison result to strong ordering:
    return std::strong_order(value, rhs.value);
  }
};
```

If possible, `std::strong_order()` yields a `std::strong_ordering` value according to the passed arguments as follows:

- Using `strong_order(`*val1*`,` *val2*`)` for the passed types if defined
- Otherwise, if the passed values are floating-point types, using the value of `totalOrder()` as specified in ISO/IEC/IEEE 60559 (for which, e.g., `-0` is less than `+0` and `-NaN` is less than any non-NAN value and `+NaN`)
- Using the new function object `std::compare_three_way{}(`*val1*`,` *val2*`)` if defined for the passed types

This is the easiest way to give floating-point types a strong ordering which even works at runtime if one or both of the operands might have the value NaN.

`std::compare_three_way` is a new function object type for calling operator <=>, just like `std::less` is a function object type for calling operator <.

For other types that have a weaker ordering and operators == and < defined, you can use the function object `std::compare_strong_order_fallback()` accordingly:

```
class Person {
  std::string name;
  SomeType value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1;       // return strong_ordering for std::string
    // map weak/partial comparison result to strong ordering:
    return std::compare_strong_order_fallback(value, rhs.value);
  }
};
```

Table *Function objects for mapping comparison category types* lists all available helper functions for mapping comparison category types.

To define `operator<=>` for a generic type, you should also consider using the function object `std::compare_three_way` or the algorithm `std::lexicographical_compare_three_way()`.

1.3 Defining operator<=> and operator==

Both `operator<=>` and `operator==` can be defined for your data types:

- Either as a member function taking one parameter
- Or as a free-standing function taking two parameters

Function Object in `std::`	**Effect**
`strong_order()`	Maps to a strong order value also for floating-point values
`weak_order()`	Maps to a weak order value also for floating-point values
`partial_order()`	Maps to a partial order value
`compare_strong_order_fallback()`	Maps to a strong order value even if only == and < are defined
`compare_weak_order_fallback()`	Maps to a weak order value even if only == and < are defined
`compare_partial_order_fallback()`	Maps to a partial order value even if only == and < are defined

Table 1.1. Function objects for mapping comparison category types

1.3.1 Defaulted `operator==` and `operator<=>`

Inside a class or data structure (as a member or `friend` function), all comparison operators can be declared as defaulted with `=default`. However, this usually only makes sense for `operator==` and `operator<=>`. The member functions have to take the second parameter as `const` lvalue reference (`const &`). Friend functions might alternatively take both parameters by value.

The defaulted operators require the support of the members and possible base classes:

- Defaulted operators == require the support of == in the members and base classes.
- Defaulted operators <=> require the support of == and either an implemented operator < or a defaulted operator <=> in the members and base classes (for details, see below).

For the generated defaulted operators, the following then applies:

- The operator is `noexcept` if comparing the members guarantees not to throw.
- The operator is `constexpr` if comparing the members is possible at compile time.

For empty classes, the defaulted operators compare all objects as equal: operators ==, <=, and >= yield `true`, operators `!=`, <, and > yield `false`, and <=> yields `std::strong_ordering::equal`.

1.3.2 Defaulted `operator<=>` Implies Defaulted `operator==`

If and only if an `operator<=>` member is defined as defaulted, then by definition a corresponding `operator==` member is also defined if no defaulted `operator==` is provided. All aspects (visibility, `virtual`, attributes, requirements, etc.) are adopted. For example:

```
template<typename T>
class Type {
  ...
 public:
  [[nodiscard]] virtual std::strong_ordering
    operator<=>(const Type&) const requires(!std::same_as<T,bool>) = default;
};
```

is equivalent to the following:

```
template<typename T>
class Type {
  ...
 public:
  [[nodiscard]] virtual std::strong_ordering
     operator<=> (const Type&) const requires(!std::same_as<T,bool>) = default;

  [[nodiscard]] virtual bool
     operator== (const Type&) const requires(!std::same_as<T,bool>) = default;
};
```

For example, the following is enough to support all six comparison operators for objects of the type `Coord`:

lang/coord.hpp

```
#include <compare>

struct Coord {
  double x{};
  double y{};
  double z{};
  auto operator<=>(const Coord&) const = default;
};
```

Note again that the member function must be `const` and that the parameter must be declared to be a `const` lvalue reference (`const &`).

You can use this data structure as follows:

lang/coord.cpp

```
#include "coord.hpp"
#include <iostream>
#include <algorithm>

int main()
{
  std::vector<Coord> coll{ {0, 5, 5}, {5, 0, 0}, {3, 5, 5},
                           {3, 0, 0}, {3, 5, 7} };

  std::sort(coll.begin(), coll.end());
  for (const auto& elem : coll) {
    std::cout << elem.x << '/' << elem.y << '/' << elem.z << '\n';
  }
}
```

The program has the following output:

```
0/5/5
3/0/0
3/5/5
3/5/7
5/0/0
```

1.3.3 Implementation of the Defaulted `operator<=>`

If `operator<=>` is defaulted and you have members or base classes and you call one of the relational operators, then the following happens:

- If `operator<=>` is defined for a member or base class, that operator is called.
- Otherwise, `operator==` and `operator<` are called to decide whether (from the point of view of the members or base classes)
  - The objects are `equal`/`equivalent` (`operator==` yields `true`)
  - The objects are `less` or `greater`
  - The objects are `unordered` (only when partial ordering is checked)

  In that case, the return type of the defaulted `operator<=>` calling these operators cannot be `auto`.

For example, consider the following declarations:

```
struct B {
  bool operator==(const B&) const;
  bool operator<(const B&) const;
};

struct D : public B {
  std::strong_ordering operator<=> (const D&) const = default;
};
```

Then:

```
D d1, d2;
d1 > d2;   // calls B::operator== and possibly B::operator<
```

If `operator==` yields `true`, we know that the result of > is `false` and that is it. Otherwise, `operator<` is called to find out whether the expression is `true` or `false`.

With

```
struct D : public B {
  std::partial_ordering operator<=> (const D&) const = default;
};
```

the compiler might even call `operator<` twice to find out whether there is any order at all.

With

```
struct B {
  bool operator==(const B&) const;
  bool operator<(const B&) const;
};
```

```
struct D : public B {
  auto operator<=> (const D&) const = default;
};
```

the compiler does not compile any call with relational operators because it cannot decide which ordering category the base class has. In that case, you need `operator<=>` in the base class too.

However, checks for equality work, because in `D`, `operator==` is automatically declared equivalent to the following:

```
struct D : public B {
  auto operator<=> (const D&) const = default;
  bool operator== (const D&) const = default;
};
```

This means that we have the following behavior:

```
D d1, d2;
d1 > d2;    // ERROR: cannot deduce comparison category of operator<=>
d1 != d2;   // OK (note: only tries operator<=> and B::operator== of a base class)
```

Equality checks always use only `operator==` of a base class (which might be generated according to a defaulted `operator<=>` though). Any `operator<` or `operator!=` in the base class is ignored.

The same applies, if `D` has a member of type `B`.

1.4 Overload Resolution with Rewritten Expressions

Let us finally elaborate on the evaluation of expressions with comparison operators with the support of rewritten calls.

Calling Equality Operators

To compile

```
x != y
```

the compiler might now try all of the following:

```
x.operator!=(y)           // calling member operator!= for x
operator!=(x, y)          // calling a free-standing operator!= for x and y

!x.operator==(y)          // calling member operator== for x
!operator==(x, y)         // calling a free-standing operator== for x and y

!x.operator==(y)          // calling member operator== generated by operator<=> for x

!y.operator==(x)          // calling member operator== generated by operator<=> for y
```

The last form is tried to support an implicit type conversion for the first operand, which requires that the operand is a parameter.

In general, the compiler tries to call:

- A free-standing operator `!=`: `operator!=(x, y)`
  or a member operator `!=`: `x.operator!=(y)`
  Having both operators `!=` defined is an ambiguity error.
- A free-standing operator ==: `!operator==(x, y)`
  or a member operator ==: `!x.operator==(y)`
  Note that the member operator == may be generated from a defaulted `operator<=>` member.
  Again, having both operators == defined is an ambiguity error. This also applies if the member `operator==` is generated due to a defaulted `operator<=>`.

When an implicit type conversion for the first operand `v` is necessary, the compiler also tries to reorder the operands. Consider:

```
42 != y       // 42 implicitly converts to the type of y
```

In that case, the compiler tries to call in that order:

- A free-standing or member operator `!=`
- A free-standing or member operator == (note that the member operator == may be generated from a defaulted `operator<=>` member)[1]

Note that a rewritten expression never tries to call a member operator `!=`.

Calling Relational Operators

For the relational operators we have similar behavior, except that the rewritten statements fall back on the new operator <=> and compare the result with `0`. The operator behaves like a three-way comparison function returning a negative value for *less*, `0` for *equal*, and a positive value for *greater* (the returned value is *not* a numeric value; it is only a value that supports the corresponding comparisons).

For example, to compile

```
x <= y
```

the compiler might now try all of the following:

```
x.operator<=(y)             // calling member operator<= for x
operator<=(x, y)            // calling a free-standing operator<= for x and y

x.operator<=>(y) <= 0       // calling member operator<=> for x
operator<=>(x, y) <= 0      // calling a free-standing operator<=> for x and y

0 <= y.operator<=>(x)       // calling member operator<=> for y
```

Again, the last form is tried to support an implicit type conversion for the first operand, for which it has to become a parameter.

[1] The original C++20 standard was fixed here slightly with http://wg21.link/p2468r2.

1.5 Using Operator <=> in Generic Code

In generic code, the new operator <=> provides some challenges. This is because there might be types that provide operator <=> and there might be types that provide some or all of the basic comparison operators.

1.5.1 `compare_three_way`

`std::compare_three_way` is a new function object type for calling operator <=>, just like `std::less` is a function object type for calling operator <.

You can use it as follows:

- To compare values of a generic type
- As a default type when you have to specify the type of a function object

For example:

```
template<typename T>
struct Value {
  T val{};
  ...
  auto operator<=> (const Value& v) const noexcept(noexcept(val<=>val)) {
      return std::compare_three_way{}(val<=>v.val);
  }
};
```

Using `std::compare_three_way` has (like `std::less`) the benefit that it even defines a total order for raw pointers (which is not the case for operators <=> or <). Therefore, you should use it when generic types are used that can be raw pointer types.

To allow programmers to forward declare `operator<=>()`, C++20 also introduces the type trait `std::compare_three_way_result` with the alias template `std::compare_three_way_result_t`:

```
template<typename T>
struct Value {
  T val{};
  ...
  std::compare_three_way_result_t<T,T>
    operator<=> (const Value& v) const noexcept(noexcept(val<=>val));
};
```

1.5.2 Algorithm `lexicographical_compare_three_way()`

To be able to compare two ranges and yield a value of the matching comparison category, C++20 also introduced the algorithm `lexicographical_compare_three_way()`. This algorithm is particularly helpful for implementing `operator<=>` for members that are collections.

For example:

lib/lexicothreeway.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

std::ostream& operator<< (std::ostream& strm, std::strong_ordering val)
{
  if (val < 0) return strm << "less";
  if (val > 0) return strm << "greater";
  return strm << "equal";
}

int main()
{
  std::vector v1{0, 8, 15, 47, 11};
  std::vector v2{0, 15, 8};

  auto r1 = std::lexicographical_compare(v1.begin(), v1.end(),
                                         v2.begin(), v2.end());

  auto r2 = std::lexicographical_compare_three_way(v1.begin(), v1.end(),
                                                   v2.begin(), v2.end());
  std::cout << "r1: " << r1 << '\n';
  std::cout << "r2: " << r2 << '\n';
}
```

The program has the following output:

```
r1: 1
r2: less
```

Note that there is no ranges support (yet) for `lexicographical_compare_three_way()`. You can neither pass a range as a single argument nor pass projection parameters:

```
auto r3 = std::ranges::lexicographical_compare(v1, v2);             // OK
auto r4 = std::ranges::lexicographical_compare_three_way(v1, v2);   // ERROR
```

1.6 Compatibility Issues with the Comparison Operators

After introducing the new rules for comparisons, it turned out that C++20 introduces some issues that may be a problem when switching from older C++ versions.

1.6.1 Delegating Free-Standing Comparison Operators

The following example demonstrates the essence of the most typical problem:

lang/spacecompat.cpp

```
#include <iostream>

class MyType {
 private:
  int value;
 public:
  MyType(int i)      // implicit constructor from int:
   : value{i} {
  }

  bool operator==(const MyType& rhs) const {
    return value == rhs.value;
  }
};

bool operator==(int i, const MyType& t) {
  return t == i;     // OK with C++17
}

int main()
{
  MyType x = 42;
  if (0 == x) {
    std::cout << "'0 == MyType{42}' works\n";
  }
}
```

We have a trivial class that stores an integral value and has an implicit constructor to initialize the object (the implicit constructor is necessary to support initialization with =):

```
class MyType {
 public:
  MyType(int i);     // implicit constructor from int
   ...
};

MyType x = 42;       // OK
```

The class also declares a member function to compare objects:

```
class MyType {
  ...
  bool operator==(const MyType& rhs) const;
  ...
};
```

However, before C++20, this does only enable implicit type conversions for the second operand. Therefore, the class or some other code introduces a global operator that swaps the order of the arguments:

```
bool operator==(int i, const MyType& t) {
  return t == i;     // OK until C++17
}
```

It would be better for the class to define the `operator==()` as a "hidden friend" (define it with `friend` inside the class structure so that both operators become parameters, you have direct access to members, and implicit type conversions are only performed if at least one parameter type fits). However, the code above had pretty much the same effect before C++20.

Unfortunately, this code no longer works in C++20.[2] It results in an endless recursion. This is because inside the global function, the expression `t == i` can also call the global `operator==()` itself, because the compiler also tries to rewrite the call as `i == t`:

```
bool operator==(int i, const MyType& t) {
  return t == i;     // finds operator==(i,t) in addition to t.operator(MyType{i})
}
```

Unfortunately, the rewritten statement is a better match because it does not need the implicit type conversion. We have no solution yet to support backward compatibility here; however, compilers are already starting to warn about code like this.

If your code only has to work with C++20, you can simply remove the free-standing function. Otherwise you have two options:

- Use an explicit conversion:

  ```
  bool operator==(int i, const MyType& t) {
    return t == MyType{i};     // OK until C++17 and with C++20
  }
  ```

- Use a feature test macro to disable the code as soon as the new feature is available:

1.6.2 Inheritance with Protected Members

For derived classes with defaulted comparison operators, there might be problems if the base class has the operators as protected members.

Note that the defaulted comparison operators require the support of comparisons in the base class. Therefore, the following does not work:

[2] Thanks to Peter Dimov and Barry Revzin for pointing out the problem and discussing it in `http://stackoverflow.com/questions/65648897`.

```
struct Base {
};

struct Child : Base {
  int i;
  bool operator==(const Child& other) const = default;
};

Child c1, c2;
...
c1 == c2;          // ERROR
```

As a consequence, you also have to provide a defaulted operator == in the base class. However, if you do not want the base class to provide operator == for the public, the obvious approach is to add a protected defaulted operator == in the base class:

```
struct Base {
 protected:
  bool operator==(const Base& other) const = default;
};

struct Child : Base {
  int i;
  bool operator==(const Child& other) const = default;
};
```

However, in this case, the defaulted comparisons for the derived class does not work. It is rejected by the current compilers implementing the specified behavior of defaulted operators that is too strict here. This hopefully will be fixed soon (see `http://wg21.link/cwg2568`).

As a workaround, you have to implement the derived operator yourself.

1.7 Afternotes

The request for implicitly defined comparison operators was first proposed by Oleg Smolsky in `http://wg21.link/n3950`. The issue was then raised again by Bjarne Stroustrup in `http://wg21.link/n4175`. However, the final proposed wording, formulated by Jens Maurer in `http://wg21.link/p0221r2`, was ultimately rejected because it was an opt-out feature (thus, existing types would automatically have comparison operators unless they declared that they did not want this). Various proposals were then considered, which Herb Sutter brought together in `http://wg21.link/p0515r0`.

The finally accepted wording was formulated by Herb Sutter, Jens Maurer, and Walter E. Brown in `http://wg21.link/p0515r3` and `http://wg21.link/p0768r1`. However, significant modifications were accepted as formulated by Barry Revzin in `http://wg21.link/p1185r2`, `http://wg21.link/p1186r3`, `http://wg21.link/p1614r2`, and `http://wg21.link/p1630r1`.

This page is intentionally left blank

Chapter 2

Placeholder Types for Function Parameters

Generic programming is a key paradigm of C++. Therefore, C++20 also provides several new features for generic programming. However, there is one basic extension that we will need more or less throughout this book: you can use now `auto` and other placeholder types to declare parameters of ordinary functions.

Later chapters will introduce more generic extensions:

- Extensions for non-type template parameters
- Lambda templates

2.1 `auto` for Parameters of Ordinary Functions

Since C++14, lambdas can use placeholders such as `auto` to declare/define their parameters:

```
auto printColl = [] (const auto& coll) {  // generic lambda
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 }
```

These placeholders allow us to pass arguments of any type, provided the operations inside the lambda are supported:

```
std::vector coll{1, 2, 4, 5};
...
printColl(coll);                     // compiles the lambda for vector<int>

printColl(std::string{"hello"});     // compiles the lambda for std::string
```

Since C++20, you can use placeholders such as `auto` for all functions (including member functions and operators):

```
void printColl(const auto& coll)    // generic function
```

```
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Such a declaration is just a shortcut for declaring a template such as the following:

```
template<typename T>
void printColl(const T& coll)       // equivalent generic function
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

The only difference is that by using `auto`, you no longer have a name for the template parameter `T`. For this reason, this feature is also called the *abbreviated function template* syntax.

Because functions with `auto` are function templates, all rules of using function templates apply. This means especially that you cannot implement a function with `auto` parameters in one translation unit (CPP file) while calling it in a different translation unit. For functions with `auto` parameters, the whole implementation belongs in a header file so that they can be used in multiple CPP files (otherwise, you have to explicitly instantiate the function in one translation unit). On the other hand, they do not need to be declared as `inline` because function templates are always inline.

In addition, you can specify the template parameters explicitly:

```
void print(auto val)
{
  std::cout << val << '\n';
}

print(64);                          // val has type int
print<char>(64);                    // val has type char
```

2.1.1 `auto` for Parameters of Member Functions

You can also use this feature to define member functions:

```
class MyType {
  ...
  void assign(const auto& newVal);
};
```

That declaration is equivalent to (with the difference that there is no type `T` defined):

```
class MyType {
  ...
  template<typename T>
  void assign(const T& newVal);
};
```

Note that templates may not be declared inside functions. With member functions using auto parameters, you can no longer define the class or data structure locally inside a function:

```
void foo()
{
  struct Data {
    void mem(auto);   // ERROR can't declare templates inside functions
  };
}
```

See sentinel1.cpp for an example of a member operator == with auto.

2.2 Using auto for Parameters in Practice

Using auto for parameters has some benefits and consequences.

2.2.1 Deferred Type Checks with auto

With auto parameters, it is significantly easier to implement code with circular dependencies.

Consider, for example, two classes that use objects of other classes. To use objects of another class, you need the definition of their type; a forward declaration is not enough (except when only declaring a reference or pointer):

```
class C2;                                   // forward declaration

class C1 {
 public:
  void foo(const C2& c2) const {            // OK
    c2.print();                             // ERROR: C2 is incomplete type
  }

  void print() const;
};

class C2 {
 public:
  void foo(const C1& c1) const {
    c1.print();                             // OK
  }
  void print() const;
};
```

While you can implement **C2**::foo() inside the class definition, you cannot implement **C1**::foo() because to check whether the call of c2.print() is valid, the compiler needs the definition of class C2.

As a consequence, you have to implement `C2::foo()` after the structures of both classes are declared:

```
class C2;

class C1 {
 public:
  void foo(const C2& c2) const;                    // forward declaration
  void print() const;
};

class C2 {
 public:
  void foo(const C1& c1) const {
    c1.print();  // OK
  }
  void print() const;
};

inline void C1::foo(const C2& c2) const {   // implementation (inline if in header)
  c2.print();    // OK
}
```

Because generic functions check members of generic parameters when the call happens, by using `auto`, you can just implement the following:

```
class C1 {
 public:
  void foo(const auto& c2) const {
    c2.print();  // OK
  }

  void print() const;
};

class C2 {
 public:
  void foo(const auto& c1) const {
    c1.print();  // OK
  }
  void print() const;
};
```

This is nothing new. You would have the same effect when declaring `C1::foo()` as a member function template. However, using `auto` makes it easier to do so.

Note that auto allows the caller to pass parameters of any type as long as the type provides a member function print(). If you do not want that, you can use the standard concept std::same_as to constrain the use of this member function for parameters of type C2 only:

```
#include <concepts>

class C2;

class C1 {
 public:
  void foo(const std::same_as<C2> auto& c2) const {
    c2.print();  // OK
  }
  void print() const;
};
...
```

For the concept, an incomplete type works fine.

2.2.2 auto Functions versus Lambdas

A function with auto parameters is different from a lambda. For example, you still cannot pass a function with auto as a parameter without specifying the generic parameter:

```
bool lessByNameFunc(const auto& c1, const auto& c2) {  // sorting criterion
  return c1.getName() < c2.getName();                   // - compare by name
}
...
std::sort(persons.begin(), persons.end(),
          lessByNameFunc);      // ERROR: can't deduce type of parameters in sorting criterion
```

Remember that the declaration of lessByName() is equivalent to:

```
template<typename T1, typename T2>
bool lessByNameFunc(const T1& c1, const T2& c2) {      // sorting criterion
  return c1.getName() < c2.getName();                   // - compare by name
}
```

Because the function template is not called directly, the compiler cannot deduce the template parameters to compile the call. Therefore, you have to specify the template parameters explicitly when passing a function template as a parameter:

```
std::sort(persons.begin(), persons.end(),
          lessByName<Customer, Customer>);  // OK
```

By using a lambda, you can just pass the lambda as it is:

```
auto lessByNameLambda = [] (const auto& c1, const auto& c2) {  // sorting criterion
                          return c1.getName() < c2.getName();   // compare by name
                        };
...
```

```
std::sort(persons.begin(), persons.end(),
          lessByNameLambda);    // OK
```

The reason is that the lambda is an object that does not have a generic type. Only the use of the object as a function is generic.

On the other hand, the explicit specification of an (abbreviated) function template parameter is easier:

- Just pass the specified type after the function name:

  ```
  void printFunc(const auto& arg) {
    ...
  }

  printFunc<std::string>("hello");    // call function template compiled for std::string
  ```

- For a generic lambda, we have to do the following:

  ```
  auto printFunc = [] (const auto& arg) {
                     ...
                   };

  printFunc.operator()<std::string>("hello");    // call lambda compiled for std::string
  ```

  For a generic lambda, the function call operator `operator()` is generic. Therefore, you have to pass the requested type as an argument to `operator()` to specify the template parameter explicitly.

2.3 `auto` for Parameters in Detail

Let us look at some aspects of abbreviated function templates in detail.

2.3.1 Basic Constraints for `auto` Parameters

Using `auto` to declare a function parameter follows the same rules as using it to declare a parameter of a lambda:

- For each parameter declared with `auto`, the function has an implicit template parameter.
- The parameters can be a parameter pack:

  ```
  void foo(auto... args);
  ```

  This is equivalent to the following (without introducing `Types`):

  ```
  template<typename... Types>
  void foo(Types... args);
  ```

- Using `decltype(auto)` is not allowed.

Abbreviated function templates can still be called with (partially) explicitly specified template parameters. The template parameters have the order of the call parameters.

For example:

```
void foo(auto x, auto y)
{
  ...
}

foo("hello", 42);                        // x has type const char*, y has type int
foo<std::string>("hello", 42);           // x has type std::string, y has type int
foo<std::string, long>("hello", 42);     // x has type std::string, y has type long
```

2.3.2 Combining Template and auto Parameters

Abbreviated function templates can still have explicitly specified template parameters. The generated template parameters for the placeholder types are added after the specified parameters:

```
template<typename T>
void foo(auto x, T y, auto z)
{
  ...
}

foo("hello", 42, '?');            // x has type const char*, T and y are int, z is char
foo<long>("hello", 42, '?');      // x has type const char*, T and y are long, z is char
```

Therefore, the following declarations are equivalent (except that you have no name for the types where auto is used):

```
template<typename T>
void foo(auto x, T y, auto z);

template<typename T, typename T2, typename T3>
void foo(T2 x, T y, T3 z);
```

As we introduce later, by using *concepts* as *type constraints*, you can constrain placeholder parameters as well as template parameters. Template parameters may then be used for such a qualification.

For example, the following declaration ensures that the second parameter y has an integral type and that the third parameter z has a type that can be converted to the type of y:

```
template<std::integral T>
void foo(auto x, T y, std::convertible_to<T> auto z)
{
  ...
}

foo(64, 65, 'c');                 // OK, x is int, T and y are int, z is char
foo(64, 65, "c");                 // ERROR: "c" cannot be converted to type int (type of 65)
foo<long,char>(64, 65, 'c');      // NOTE: x is char, T and y are long, z is char
```

Note that the last statement specifies the type of the parameters in the wrong order.

The fact that the order of template parameters is not as expected might lead to subtle bugs. Consider the following example:

lang/tmplauto.cpp

```
#include <vector>
#include <ranges>

void addValInto(const auto& val, auto& coll)
{
  coll.insert(val);
}

template<typename Coll>     // Note: different order of template parameters
requires std::ranges::random_access_range<Coll>
void addValInto(const auto& val, Coll& coll)
{
  coll.push_back(val);
}

int main()
{
  std::vector<int> coll;
  addValInto(42, coll);     // ERROR: ambiguous
}
```

Due to using `auto` only for the first parameter in the second declaration of `addValInto()`, the order of the template parameters differs. Due to http://wg21.link/p2113r0, which was accepted for C++20, this means that overload resolution does not prefer the second declaration over the first one and we get an ambiguity error.[1]

For this reason, be careful when mixing template and `auto` parameters. Ideally, make the declarations consistent.

2.4 Afternotes

`auto` for parameters of ordinary functions was first proposed together with the option to use type constraints for them by Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann, and Erich Keane in http://wg21.link/p1141r0. The finally accepted wording was formulated by Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann, Erich Keane, Walter E. Brown, Michael Spertus, and Richard Smith in http://wg21.link/p1141r2.

[1] Not all compilers handle this correctly, yet.

Chapter 3

Concepts, Requirements, and Constraints

This chapter introduces the new C++ feature *concepts*, with the keywords `concept` and `requires`. The feature is used to constrain the availability of generic code according to specified requirements.

For sure, *concepts* are a milestone for C++, because concepts provide a language feature for something we need a lot when writing generic code: specifying *requirements*. Although there have been workarounds, we now have an easy and readable way to specify the requirements for generic code, get better diagnostics when requirements are broken, disable generic code when it does not work (even though it might compile), and switch between different generic code for different types.

3.1 Motivating Example of Concepts and Requirements

Consider the following function template that returns the maximum of two values:

```
template<typename T>
T maxValue(T a, T b) {
  return b < a ? a : b;
}
```

This function template can be called for two arguments that have the same type, provided the operations performed for the parameters (comparing with `operator<` and copying) are valid.

However, passing two pointers would compare their addresses instead of the values they refer to.

3.1.1 Improving the Template Step by Step

Using a `requires` Clause

To fix that, we can equip the template with a ***constraint*** so that it is not available if raw pointers are passed:

```
template<typename T>
requires (!std::is_pointer_v<T>)
```

```
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

Here, the constraint is formulated in a ***requires clause***, which is introduced with the keyword `requires` (there are other ways to formulate constraints).

To specify the constraint that the template cannot be used for raw pointers, we use the standard *type trait* `std::is_pointer_v<>` (which yields the `value` member of the standard type trait `std::is_pointer<>`).[1] With this constraint, we can no longer use the function template for raw pointers:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';        // OK: maximum value of ints
std::cout << maxValue(&x, &y) << '\n';      // ERROR: constraint not met
```

The requirement is a compile-time check and has no impact on the performance of the compiled code. It merely means that the template cannot be used for raw pointers. When raw pointers are passed, the compiler behaves as if the template were not there.

Defining and Using a `concept`

Probably, we will need a constraint for pointers more than once. For this reason, we can introduce a ***concept*** for the constraint:

```
template<typename T>
concept IsPointer = std::is_pointer_v<T>;
```

A *concept* is a template that introduces a name for one or more requirements that apply to the passed template parameters so that we can use these requirements as constraints. After the equal sign (you cannot use braces here), we have to specify the requirements as a Boolean expression that is evaluated at compile time. In this case, we require that the template argument used to specialize `IsPointer<>` has to be a raw pointer.

We can use this concept to constrain the `maxValue()` template as follows:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

[1] Type traits were introduced as standard type functions with C++11 and the ability to call them with an `_v` suffix was introduced with C++17.

Overloading with Concepts

By using constraints and concepts, we can even overload the `maxValue()` template to have one implementation for pointers and one for other types:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)            // maxValue() for non-pointers
{
  return b < a ? a : b;         // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)         // maxValue() for pointers
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

Note that `requires` clauses that just constrain a template with a concept (or multiple concepts combined with `&&`) no longer need parentheses. A negated concept always needs parentheses.

We now have two function templates with the same name, but only one of them is available for each type:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';      // calls maxValue() for non-pointers
std::cout << maxValue(&x, &y) << '\n';    // calls maxValue() for pointers
```

Because the implementation for pointers delegates the computations of the return value to the objects the pointers refer to, the second call uses both `maxValue()` templates. When passing pointers to `int`, we instantiate the template for pointers with `T` as `int*`, and the basic `maxValue()` template that is for non-pointers with `T` as `int`.

This even works recursively now. We can ask for the maximum value of a pointer to a pointer to an `int`:

```
int* xp = &x;
int* yp = &y;
std::cout << maxValue(&xp, &yp) << '\n';  // calls maxValue() for int**
```

Overload Resolution with Concepts

Overload resolution considers templates with constraints as more specialized than templates without constraints. Therefore, it is enough to constrain the implementation only for pointers:

```
template<typename T>
T maxValue(T a, T b)            // maxValue() for a value of type T
{
  return b < a ? a : b;         // compare values
}
```

```
template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)        // maxValue() for pointers (higher priority)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

However, be careful: overloading once using references and once using non-references might cause ambiguities.

By using concepts, we can even prefer some constraints over others. However, this requires the use of concepts that *subsume* other concepts.

Type Constraints

If a constraint is a single concept that is applied to a parameter, there are ways to shortcut the specification of the constraint. First, you can specify it directly as a ***type constraint*** when declaring the template parameter:

```
template<IsPointer T>    // only for pointers
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

In addition, you can use the concept as a *type constraint* when declaring parameters with `auto`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

This also works for parameters passed by reference:

```
auto maxValue3(const IsPointer auto& a, const IsPointer auto& b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

Note that by constraining both parameters directly, we changed the specification of the template: we no longer require that `a` and `b` have to have the same type. We require only that both are pointer-like objects of an arbitrary type.

When using the template syntax, the equivalent code looks as follows:

```
template<IsPointer T1, IsPointer T2>    // only for pointers
auto maxValue(T1 a, T2 b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

We should probably also allow different types for the basic function template that compares the values. One way to do that is to specify two template parameters:

```
template<typename T1, typename T2>
auto maxValue(T1 a, T2 b)      // maxValue() for values
{
  return b < a ? a : b;        // compare values
}
```

The other option would be to also use `auto` parameters:

```
auto maxValue(auto a, auto b)  // maxValue() for values
{
  return b < a ? a : b;        // compare values
}
```

We could now pass a pointer to an `int` and a pointer to a `double`.

Trailing `requires` Clauses

Consider the pointer version of `maxValue()`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

There is still an implicit requirement that is not obvious: after dereferencing, the values have to be comparable.

Compilers detect that requirement when (recursively) instantiating the `maxValue()` templates. However, the error message might be a problem because the error occurs late and the requirement is not visible in the declaration of `maxValue()` for pointers.

To let the pointer version directly require in its declaration that the values the pointers point to have to be comparable, we can add another constraint to the function:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires IsComparableWith<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

Here, we use a ***trailing requires clause***, which can be specified ***after*** the parameter list. It has the benefit that it can use the name of a parameter or combine even multiple parameter names to formulate constraints.

Standard Concepts

In the previous example, we did not define the concept `IsComparableWith`. We could do that using a requires expression (which we introduce in a moment); however, we could also use a concept of the C++ standard library, For example, we could declare the following:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::totally_ordered_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

The concept `std::totally_ordered_with` takes two template parameters to check whether the values of the passed types are comparable with the operators ==, !=, <, <=, >, and >=.

The standard library has many standard concepts for common constraints. They are provided in the namespace `std` (sometimes a subnamespace is used).

For example, we could also use the concept `std::three_way_comparable_with`, which in addition requires that the new operator <=> (which gives the concept the name) is supported. To check support for comparisons of two objects of the same type, we can use the concept `std::totally_ordered`.

`requires` Expressions

So far, the `maxValue()` templates do not work for pointer-like types that are not raw pointers, such as smart pointers. If the code should also compile for those types, we should better define that pointers are objects for which we can call `operator*`.

Such a requirement is easy to specify since C++20:

```
template<typename T>
concept IsPointer = requires(T p) { *p; };     // expression *p has to be well-formed
```

Instead of using the type trait for raw pointers, the concept formulates a simple requirement: the expression `*p` has to be valid for an object `p` of the passed type `T`.

Here, we are using the `requires` keyword again to introduce a ***requires expression***, which can define one or more ***requirements*** for types and parameters. By declaring a parameter `p` of type `T`, we can simply specify which operations for such an object have to be supported.

We can also require multiple operations, type members, and that expressions yield constrained types. For example:

```
template<typename T>
concept IsPointer = requires(T p) {
                      *p;                               // operator * has to be valid
                      p == nullptr;                     // can compare with nullptr
                      {p < p} -> std::same_as<bool>;    // operator < yields bool
                    };
```

We specify three requirements, which all apply to a parameter `p` of the type `T` that we define this concept for:

- The type has to support the operator `*`.
- The type has to support the operator <, which has to yield type `bool`.
- Objects of that type have to be comparable with `nullptr`.

Note that we do not need two parameters of type `T` to check whether `<` can be called. The runtime value does not matter. However, note that there are some restrictions for how to specify what an expression yields (e.g., you cannot specify just `bool` without `std::same_as<>` there).

Note also that we do ***not*** require `p` to be a pointer that ***is*** equal to `nullptr` here. We require only that we ***can*** compare `p` with `nullptr`. However, that rules out iterators, because in general, they cannot be compared with `nullptr` (except when they happen to be implemented as raw pointers, which, for example, is typically the case for type `std::array<>`).

Again, this is a compile-time constraint that has no impact on the generated code; we only decide for which types the code compiles. Therefore, it does not matter whether we declare the parameter `p` as a value or as a reference.

You could also use the *requires expression* as an ad-hoc constraint directly in the *requires clause* (which looks a bit funny but makes total sense once you understand the difference between a requires clause and a requires expression and that both need the keyword `requires`):

```cpp
template<typename T>
requires requires(T p) { *p; }  // constrain template with ad-hoc requirement
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);
}
```

3.1.2 A Complete Example with Concepts

We have now introduced everything necessary to look at a complete example program for concepts that computes the maximum value for plain values and pointer-like objects:

lang/maxvalue.cpp

```cpp
#include <iostream>

// concept for pointer-like objects:
template<typename T>
concept IsPointer = requires(T p) {
                      *p;            // operator * has to be valid
                      p == nullptr;  // can compare with nullptr
                      {p < p} -> std::convertible_to<bool>;  // < yields bool
                    };

// maxValue() for plain values:
auto maxValue(auto a, auto b)
{
  return b < a ? a : b;
}
```

```
// maxValue() for pointers:
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::totally_ordered_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);   // return maximum value of where the pointers refer to
}

int main()
{
  int x = 42;
  int y = 77;
  std::cout << maxValue(x, y) << '\n';        // maximum value of ints
  std::cout << maxValue(&x, &y) << '\n';      // maximum value of where the pointers point to

  int* xp = &x;
  int* yp = &y;
  std::cout << maxValue(&xp, &yp) << '\n';    // maximum value of pointer to pointer

  double d = 49.9;
  std::cout << maxValue(xp, &d) << '\n';      // maximum value of int and double pointer
}
```

Note that we cannot use `maxValue()` to check for the maximum of two iterator values:

```
std::vector coll{0, 8, 15, 11, 47};
auto pos = std::find(coll.begin(), coll.end(), 11);    // find specific value
if (pos != coll.end()) {
  // maximum of first and found value:
  auto val = maxValue(coll.begin(), pos);               // ERROR
}
```

The reason is that we require the parameters to be comparable with `nullptr`, which is not required to be supported by iterators. Whether or not this is what you want is a design question. However, this example demonstrates that it is important to think carefully about the definition of general concepts.

3.2 Where Constraints and Concepts Can Be Used

You can use requires clauses or concepts to constrain almost all forms of generic code:

- Function templates:

```
template<typename T>
requires ...
void print(const T&) {
  ...
}
```

- Class templates
  ```
  template<typename T>
  requires ...
  class MyType {
    ...
  }
  ```
- Alias templates
- Variable templates
- You can even constrain member functions

For these templates, you can constrain both type and value parameters.

However, note that you cannot constrain concepts:

```
template<std::ranges::sized_range T>                  // ERROR
concept IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

Instead, you have to specify this as follows:

```
template<typename T>
concept IsIntegralValType = std::ranges::sized_range<T> &&
                            std::integral<std::ranges::range_value_t<T>>;
```

3.2.1 Constraining Alias Templates

Here is an example of constraining alias templates (generic using declarations):

```
template<std::ranges::range T>
using ValueType = std::ranges::range_value_t<T>;
```

The declaration is equivalent to the following:

```
template<typename T>
requires std::ranges::range<T>
using ValueType = std::ranges::range_value_t<T>;
```

Type ValueType<> is now defined only for types that are ranges:

```
ValueType<int> vt1;                         // ERROR
ValueType<std::vector<int>> vt2;            // int
ValueType<std::list<double>> vt3;           // double
```

3.2.2 Constraining Variable Templates

Here is an example of constraining variable templates:

```
template<std::ranges::range T>
constexpr bool IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

Again, this is equivalent to the following:

```
template<typename T>
requires std::ranges::range<T>
constexpr bool IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

The Boolean variable template is now defined only for ranges:

```
bool b1 = IsIntegralValType<int>;              // ERROR
bool b2 = IsIntegralValType<std::vector<int>>;  // true
bool b3 = IsIntegralValType<std::list<double>>; // false
```

3.2.3 Constraining Member Functions

Requires clauses can also be part of the declaration of member functions. That way, programmers can specify different APIs based on requirements and concepts.

Consider the following example:

lang/valorcoll.hpp

```
#include <iostream>
#include <ranges>

template<typename T>
class ValOrColl {
  T value;
 public:
  ValOrColl(const T& val)
   : value{val} {
  }
  ValOrColl(T&& val)
   : value{std::move(val)} {
  }

  void print() const {
    std::cout << value << '\n';
  }

  void print() const requires std::ranges::range<T> {
    for (const auto& elem : value) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
};
```

Here, we define a class `ValOrColl` that can hold a single value or a collection of values as `value` of type `T`. Two `print()` member functions are provided and the class uses the standard concept `std::ranges::range` to decided which one to call:

- If type `T` is a collection, the constraint is satisfied so that both `print()` member functions are available. However, the second `print()`, which iterates over the elements of the collection, is preferred by overload resolution, because this member function has a constraint.
- If type `T` is not a collection, only the first `print()` is available and therefore used.

For example, you can use the class as follows:

lang/valorcoll.cpp

```
#include "valorcoll.hpp"
#include <vector>

int main()
{
  ValOrColl o1 = 42;
  o1.print();

  ValOrColl o2 = std::vector{1, 2, 3, 4};
  o2.print();
}
```

The program has the following output:

```
42
1 2 3 4
```

Note that you can constrain only templates this way. You cannot use `requires` to constrain ordinary functions:

```
void foo() requires std::numeric_limits<char>::is_signed    // ERROR
{
  ...
}
```

One example of a constraining member function inside the C++standard library is the conditional availability of `begin()` for const views.

3.2.4 Constraining Non-Type Template Parameters

It is not only types that you can constrain. You can also constrain values that are template parameters (non-type template parameters (NTTP)). For example:

```
template<int Val>
concept LessThan10 = Val < 10;
```

Or more generic:

```
template<auto Val>
concept LessThan10 = Val < 10;
```

This concept can be used as follows:

```
template<typename T, int Size>
requires LessThan10<Size>
class MyType {
  ...
};
```

We will discuss more examples later.

3.3 Typical Applications of Concepts and Constraints in Practice

There are multiple reasons why using requirements as constraints can be useful:

- Constraints help us to *understand* the restrictions on templates and to get more understandable error messages when requirements are broken.
- Constraints can be used to *disable generic code* for cases where the code does not make sense:
  - For some types, generic code might compile but not do the right thing.
  - We might have to fix overload resolution, which decides which operation to call if there are multiple valid options.
- Constraints can be used to *overload* or *specialize* generic code so that different code is compiled for different types.

Let us take a closer look at these reasons by developing another example step by step. This way, we can also introduce a couple of additional details about constraints, requirements, and concepts.

3.3.1 Using Concepts to Understand Code and Error Messages

Assume we want to write generic code that inserts the value of an object into a collection. Thanks to templates, we can implement it once as generic code that is compiled for the types of passed objects once we know them:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

This code does not always compile. There is an implicit requirement for the types of the passed arguments: the call to `push_back()` for the value of type `T` has to be supported for the container of type `Coll`.

You could also argue that this is a combination of multiple basic requirements:

- Type `Coll` has to support `push_back()`.
- There has to be a conversion from type `T` to the type of the elements of `Coll`.
- If the passed argument has the element type of `Coll`, that type has to support copying (because a new element is initialized with the passed value).

If any of these requirements are broken, the code does not compile. For example:

```
std::vector<int> vec;
add(vec, 42);        // OK
add(vec, "hello");   // ERROR: no conversion from string literal to int

std::set<int> coll;
add(coll, 42);       // ERROR: no push_back() supported by std::set<>

std::vector<std::atomic<int>> aiVec;
std::atomic<int> ai{42};
add(aiVec, ai);      // ERROR: cannot copy/move atomics
```

When compilation fails, error messages can be very clear, such as when the member `push_back()` is not found on the top level of a template:

```
prog.cpp: In instantiation of 'void add(Coll&, const T&)
          [with Coll = std::__debug::set<int>; T = int]':
prog.cpp:17:18:   required from here
prog.cpp:11:8: error: 'class std::set<int>' has no member named 'push_back'
   11 |    coll.push_back(val);
      |    ~~~~~~~~~~~~~~
```

However, generic error messages can also be very tough to read and understand. For example, when the compiler has to deal with the broken requirement that copying is not supported, the problem is detected in the deep darkness of the implementation of `std::vector<>`. We get between 40 and 90 lines of error messages where we have to look carefully for the broken requirement:

```
...
prog.cpp:11:17:   required from 'void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]'
prog.cpp:25:18:   required from here
.../include/bits/stl_construct.h:96:17:
  error: use of deleted function
         'std::atomic<int>::atomic(const std::atomic<int>&)'
   96 |      -> decltype(::new((void*)0) _Tp(std::declval<_Args>()...))
      |                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

You might think that you can improve the situation by defining and using a concept that checks whether you can perform the `push_back()` call:

```cpp
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
    c.push_back(v);
};

template<typename Coll, typename T>
requires SupportsPushBack<Coll, T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

The error message for not finding `push_back()` might now be as follows:

```
prog.cpp:27:4: error: no matching function for call to 'add(std::set<int>&, int)'
   27 | add(coll, 42);
      | ~~~~~~~~~~~~~
prog.cpp:14:6: note: candidate: 'template<class Coll, class T>  requires  ...'
   14 | void add(Coll& coll, const T& val)
      |      ^~~
prog.cpp:14:6: note:   template argument deduction/substitution failed:
prog.cpp:14:6: note: constraints not satisfied
prog.cpp: In substitution of 'template<class Coll, class T>  requires  ...
          [with Coll = std::set<int>; T = int]':
```

```
prog.cpp:27:4:   required from here
prog.cpp:8:9:   required for the satisfaction of 'SupportsPushBack<Coll, T>'
                [with Coll = std::set<int, std::less<int>, std::allocator<int> >; T = int]
prog.cpp:8:28:   in requirements with 'Coll c', 'T v'
                 [with T = int; Coll = std::set<int, std::less<int>, std::allocator<int> >]
prog.cpp:9:16: note: the required expression 'c.push_back(v)' is invalid
    9 |     c.push_back(v);
      |     ~~~~~~~~~~~~~~
```

However, when passing atomics, the check for being copyable is still detected in the deep darkness of the code for `std::vector<>` (this time when the concept is checked, instead of when the code is compiled).

In that case, the situation improves when we specify the basic constraint for the use of `push_back()` as a requirement instead:

```
template<typename Coll, typename T>
requires std::convertible_to<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

Here, we use the standard concept `std::convertible_to` to require that the type of the passed argument `T` can be (implicitly or explicitly) converted to the element type of the collection.

Now, if the requirement is broken, we can get an error message with the broken concept and the location where it is broken. For example:[2]

```
...
prog.cpp:11:17:   In substitution of 'template<class Coll, class T>
                  requires  convertible_to<T, typename Coll::value_type>
                  void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]':
prog.cpp:25:18:   required from here
.../include/concepts:72:13: required for the satisfaction of
                  'convertible_to<T, typename Coll::value_type>
                  [with T = std::atomic<int>;
                        Coll = std::vector<std::atomic<int>,
                                          std::allocator<std::atomic<int> > >]'
.../include/concepts:72:30: note: the expression 'is_convertible_v<_From, _To>
                  [with _From = std::atomic<int>; _To = std::atomic<int>]'
                  evaluated to 'false'
   72 |     concept convertible_to = is_convertible_v<_From, _To>
      |                              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

See rangessort.cpp for another example of algorithms that check the constraints of their parameters.

[2] This is just one example of a possible message, not necessarily matching the situation of a specific compiler.

3.3.2 Using Concepts to Disable Generic Code

Assume we want to provide a special implementation for an `add()` function template like the one introduced above. When dealing with floating-point values, something different or extra should happen.

A naive approach might be to overload the function template for `double`:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)      // for generic value types
{
  coll.push_back(val);
}

template<typename Coll>
void add(Coll& coll, double val)        // for floating-point value types
{
  ...   // special code for floating-point values
  coll.push_back(val);
}
```

As expected, when we pass a `double` as a second argument, the second function is called; otherwise, the generic argument is used:

```
std::vector<int> iVec;
add(iVec, 42);          // OK: calls add() for T being int

std::vector<double> dVec;
add(dVec, 0.7);         // OK: calls 2nd add() for double
```

When passing a `double`, both function overloads match. The second overload is preferred because it is a perfect match for the second argument.

However, if we pass a `float`, we have the following effect:

```
float f = 0.7;
add(dVec, f);           // OOPS: calls 1st add() for T being float
```

The reason lies in the sometimes subtle details of overload resolution. Again, both functions could be called. Overload resolution has some general rules, such as:

- Calls with no type conversion are preferred over calls having a type conversion.
- Calls of ordinary functions are preferred over calls of function templates.

Here, however, overload resolution has to decide between a call with a type conversion and a call of a function template. By rule, in that case, the version with the template parameter is preferred.

Fixing Overload Resolution

The fix for the wrong overload resolution is pretty simple. Instead of declaring the second parameter with a specific type, we should just require that the value to be inserted has a floating-point type. For this, we can constrain the function template for floating-point values by using the new standard concept `std::floating_point`:

```
template<typename Coll, typename T>
requires std::floating_point<T>
void add(Coll& coll, const T& val)
{
  ...   // special code for floating-point values
  coll.push_back(val);
}
```

Because we use a concept that applies to a single template parameter we can also use the shorthand notation:

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)
{
  ...   // special code for floating-point values
  coll.push_back(val);
}
```

Alternatively, we can use `auto` parameters:

```
void add(auto& coll, const std::floating_point auto& val)
{
  ...   // special code for floating-point values
  coll.push_back(val);
}
```

For `add()`, we now have two function templates that can be called: one without and one with a specific requirement:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)      // for generic value types
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)      // for floating-point value types
{
  ...   // special code for floating-point values
  coll.push_back(val);
}
```

This is enough because overload resolution also prefers overloads or specializations that have constraints over those that have fewer or no constraints:

```
std::vector<int> iVec;
add(iVec, 42);          // OK: calls add() for generic value types

std::vector<double> dVec;
add(dVec, 0.7);         // OK: calls add() for floating-point types
```

Do Not Differ More Than Necessary

If two overloads or specializations have constraints, it is important that overload resolution can decide which one is better. To support this, the signatures should not differ more than necessary.

If the signatures differ too much, the more constrained overload might not be preferred. For example, if we declare the overload for floating-point values to take the argument by value, passing a floating-point value becomes ambiguous:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)   // note: pass by const reference
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, T val)          // note: pass by value
{
  ...  // special code for floating-point values
  coll.push_back(val);
}

std::vector<double> dVec;
add(dVec, 0.7);       // ERROR: both templates match and no preference
```

The latter declaration is no longer a special case of the former declaration. We just have two different function templates that could both be called.

If you really want to have different signatures, you have to constrain the first function template not to be available for floating-point values.

Restrictions on Narrowing

We have another interesting issue in this example: both function templates allow us to pass a `double` to add it into a collection of `int`:

```
std::vector<int> iVec;

add(iVec, 1.9);       // OOPS: add 1
```

The reason is that we have implicit type conversions from `double` to `int` (due to being compatible with the programming language C). Such an implicit conversion where we might lose parts of the value is called *narrowing*. It means that the code above compiles and converts the value `1.9` to `1` before it is inserted.

If you do not want to support narrowing, you have multiple options. One option is to disable type conversions completely by requiring that the passed value type matches the element type of the collection:

```
requires std::same_as<typename Coll::value_type, T>
```

However, this would also disable useful and safe type conversions.

For that reason, it is better to define a concept that yields whether a type can be converted to another type without narrowing, which is possible in a short tricky requirement:[3]

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
         -> std::same_as<To[1]>;
    };
```

We can then use this concept to formulate a corresponding constraint:

```
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  ...
}
```

Subsuming Constraints

It would probably be enough to define the concept for narrowing conversions above without requiring the concept `std::convertible_to`, because the rest checks this implicitly:

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing = requires (From&& x) {
    { std::type_identity_t<To[]>{std::forward<From>(x)} } -> std::same_as<To[1]>;
};
```

However, there is an important benefit if the concept `ConvertsWithoutNarrowing` also checks for the concept `std::convertible_to`. In that case, the compiler can detect that `ConvertsWithoutNarrowing` is more constrained than `std::convertible_to`. The terminology is that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`.

That allows a programmer to do the following:

```
template<typename F, typename T>
requires std::convertible_to<F, T>
void foo(F, T)
{
   std::cout << "may be narrowing\n";
}
```

[3] Thanks to Giuseppe D'Angelo and Zhihao Yuan for the trick to check for narrowing conversions introduced in `http://wg21.link/p0870`.

```
template<typename F, typename T>
requires ConvertsWithoutNarrowing<F, T>
void foo(F, T)
{
   std::cout << "without narrowing\n";
}
```

Without specifying that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`, the compiler would raise an ambiguity error here when calling `foo()` with two parameters that convert to each other without narrowing.

In the same way, concepts can subsume other concepts, which means that they count as more specialized for overload resolution. In fact, the C++ standard concepts build a pretty complex subsumption graph.

We will discuss details of subsumption later.

3.3.3 Using Requirements to Call Different Functions

Finally, we should make our `add()` function template more flexible:

- We might also want to support collections that provide only `insert()` instead of `push_back()` for inserting new elements.
- We might want to support passing a collection (container or range) to insert multiple values.

Yes, you can argue that these are different functions and that they should have different names and if that works, it is often better to use different names. However, the C++ standard library is a good example of the benefits you can get if different APIs are harmonized. For example, you can use the same generic code to iterate over all containers, although internally, containers use very different ways to go to the next element and access its value.

Using Concepts to Call Different Functions

Having just introduced concepts, the “obvious” approach might be to introduce a concept to find out whether a certain function call is supported:

```
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
    c.push_back(v);
};
```

Note that we can also define a concept that needs only the collection as a template parameter:

```
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};
```

Note that we do not have to use `typename` here to use `Coll::value_type`. Since C++20, `typename` is no longer required when it is clear by the context that a qualified member must be a type.

There are various other ways to declare this concept:

- You can use `std::declval<>()` to get a value of the element type:

  ```
  template<typename Coll>
  concept SupportsPushBack = requires(Coll coll) {
    coll.push_back(std::declval<typename Coll::value_type&>());
  };
  ```

  Here, you can see that the definition of concepts and requirements does not create code. It is an unevaluated context where we can use `std::declval<>()` for “assume we have an object of this type” and it does not matter whether we declare `coll` as a value or as a non-`const` reference.

  Note that the `&` is important here. Without `&`, we would only require that we can insert an rvalue (such as a temporary object) using move semantics. With `&`, we create an lvalue so that we require that `push_back()` copies.[4]

- You could use `std::ranges::range_value_t` instead of the `value_type` member:

  ```
  template<typename Coll>
  concept SupportsPushBack = requires(Coll coll) {
    coll.push_back(std::declval<std::ranges::range_value_t<Coll>>());
  };
  ```

  In general, using `std::ranges::range_value_t<>` makes code more generic when the element type of a collection is needed (e.g., it also works with raw array). However, because we require the member `push_back()` here, also requiring the member `value_type` does not hurt.

With the concept `SupportPushBack` for one parameter, we can provide two implementations:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}

template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.insert(val);
}
```

In this case, we not need a named requirement `SupportsInsert` here because the `add()` with the additional requirement is more special, which means that overload resolution prefers it. However, there are only a few containers that support calling `insert()` with only one argument. To avoid problems with other overloads and calls of `add()`, we should probably better also have a constraint here.

Because we define the requirement as a concept, we can even use it as a *type constraint* for the template parameter:

[4] Thanks to Daniel Krügler for pointing this out.

```
template<SupportsPushBack Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

As a concept, we can also use it as a type constraint for `auto` as parameter types:

```
void add(SupportsPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

template<typename Coll, typename T>
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}
```

Concepts for `if constexpr`

We can also use the concept `SupportsPushBack` directly in an `if constexpr` condition:

```
if constexpr (SupportsPushBack<decltype(coll)>) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

Combining `requires` with `if constexpr`

We can even skip introducing a concept and pass the `requires` expression directly as a condition to the compile-time `if`:[5]

```
if constexpr (requires { coll.push_back(val); }) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

This is a nice way to switch between two different function calls in generic code. It is especially recommended when introducing a concept is not worthwhile.

[5] Thanks to Arthur O'Dwyer for pointing this out.

Concepts versus Variable Templates

You might wonder why using concepts is better than using a variable template of type bool (like type traits do) such as the following:

```
template<typename T>
constexpr bool SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};
```

Concepts have the following benefits:

- They subsume.
- They can be used as type constraints directly in front of template parameters or auto.
- They can be used with a compile-time if when using ad-hoc requirements.

If you do not need any of these benefits, the question of whether to prefer the definition of a concept or a variable template of type bool becomes interesting and this is discussed later in detail.

Inserting Single and Multiple Values

To provide an overload that deals with multiple values passed as one collection, we can simply add constraints for them. The standard concept std::ranges::input_range can be used for that:

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), val.begin(), val.end());
}

template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(val.begin(), val.end());
}
```

Again, as long as the overload has this as an additional constraint, these functions will be preferred.

The concept std::ranges::input_range is a concept introduced to deal with ranges, which are collections you can iterate over with begin() and end(). However, ranges are not required to have begin() and end() as member functions. Code that deals with ranges should therefore use the helpers std::ranges::begin() and std::ranges::end() that the ranges library provides:

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), std::ranges::begin(val), std::ranges::end(val));
}

template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
```

```
{
  coll.insert(std::ranges::begin(val), std::ranges::end(val));
}
```

These helpers are function objects so that using them avoids ADL problems.

Dealing with Multiple Constraints

By bringing together all useful concepts and requirements, we could place them all in one function at different locations.

```
template<SupportsPushBack Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
              std::ranges::begin(val), std::ranges::end(val));
}
```

To disable narrowing conversions, we use `std::ranges::range_value_t` to pass the element type of the ranges to `ConvertsWithoutNarrowing`. `std::ranges::range_value_t` is another ranges utility for getting the element type of ranges when iterating over them.

We could also formulate them together in the requires clause:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll> &&
         std::ranges::input_range<T> &&
         ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
              std::ranges::begin(val), std::ranges::end(val));
}
```

Both ways of declaring the function template are equivalent.

3.3.4 The Example as a Whole

The previous subsections provided a huge amount of flexibility. So, let us bring all options together to have at least one full example as a whole:

lang/add.cpp

```
#include <iostream>
#include <vector>
#include <set>
#include <ranges>
#include <atomic>

// concept for container with push_back():
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};

// concept to disable narrowing conversions:
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
          -> std::same_as<To[1]>;
    };


// add() for single value:
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}

// add() for multiple values:
template<typename Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
```

```
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.insert(coll.end(),
                std::ranges::begin(val), std::ranges::end(val));
  }
  else {
    coll.insert(std::ranges::begin(val), std::ranges::end(val));
  }
}

int main()
{
  std::vector<int> iVec;
  add(iVec, 42);         // OK: calls push_back() for T being int

  std::set<int> iSet;
  add(iSet, 42);         // OK: calls insert() for T being int

  short s = 42;
  add(iVec, s);          // OK: calls push_back() for T being short

  long long ll = 42;
  //add(iVec, ll);      // ERROR: narrowing
  //add(iVec, 7.7);     // ERROR: narrowing

  std::vector<double> dVec;
  add(dVec, 0.7);        // OK: calls push_back() for floating-point types
  add(dVec, 0.7f);       // OK: calls push_back() for floating-point types
  //add(dVec, 7);       // ERROR: narrowing

  // insert collections:
  add(iVec, iSet);       // OK: insert set elements into a vector
  add(iSet, iVec);       // OK: insert vector elements into a set

  // can even insert raw array:
  int vals[] = {0, 8, 18};
  add(iVec, vals);       // OK
  //add(dVec, vals);    // ERROR: narrowing
}
```

As you can see, I decided to use the concept `SupportsPushBack` for one template parameter inside the various function templates with `if constexpr`.

3.3.5 Former Workarounds

Constraining templates was already possible before C++20. However, the approaches for doing this were often not easy to use and could provide significant drawbacks.

SFINAE

The major approach for disabling the availability of templates before C++20 was SFINAE. The term "SFINAE" (pronounced like *sfee-nay*) stands for "*substitution failure is not an error*" and means the rule that we simply ignore generic code if its declaration is not well-formed instead of raising a compile-time error.

For example, to switch between `push_back()` and `insert()`, we could declare the function templates before C++20 as follows:

```
template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.push_back(val))
{
  return coll.push_back(val);
}

template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.insert(val))
{
  return coll.insert(val);
}
```

Because we make the call to `push_back()` here part of the first template **declaration**, this template is ignored if `push_back()` is not supported. The corresponding declaration is necessary in the second template for `insert()` (overload resolution does ignore return types and would complain if both function templates can be used).

However, that is a very subtle way to place a requirement and programmers might overlook it easily.

`std::enable_if<>`

For more complicated cases of disabling generic code, since C++11, the C++ standard library provides `std::enable_if<>`.

This is a type trait that takes a Boolean condition that yields invalid code if `false`. By using the `std::enable_if<>` somewhere inside a declaration, this could also "SFINAE out" generic code.

For example, you could use the `std::enable_if<>` type trait to exclude types from calling the `add()` function template in the following way:

```
// disable the template for floating-point values:
template<typename Coll, typename T,
         typename = std::enable_if_t<!std::is_floating_point_v<T>>>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

The trick is to insert an additional template parameter to be able to use the trait `std::enable_if<>`. The trait yields `void` if it does not disable the template (you can specify that it yields another type as an optional second template parameter).

Code like this is also hard to write and read and has subtle drawbacks. For example, to provide another overload of `add()` with a different constraint, you would need yet another template parameter. Otherwise, you would have two different overloads of the same function because for the compiler `std::enable_if<>` does not change the signature. In addition, you have to take care that for each call exactly one of the different overloads is available.

Concepts provide a far more readable way of formulating constraints. This includes that overloads of a template with different constraints do not violate the one definition rule as long as only one constraint is met and that even multiple constraints can be met provided one constraint subsumes another constraint.

3.4 Semantic Constraints

Concepts might check both syntactic and semantic constraints:

- **Syntactic constraints** mean that at compile time, we can check whether certain functional requirements are satisfied ("*Is a specific operation supported?*" or "*Does a specific operation yield a specific type?*").
- **Semantic constraints** mean that certain requirements are satisfied that can only be checked at runtime ("*Does an operation have the same effect?*" or "*Does the same operation performed for a specific value always yield the same result?*").

Sometimes, concepts allow programmers to convert semantic constraints into syntactic constraints by providing an interface to specify that a semantic constraint is fulfilled or is not fulfilled.

3.4.1 Examples of Semantic Constraints

Let us look at some examples of semantic constraints.

`std::ranges::sized_range`

An example of a semantic constraint is the concept `std::ranges::sized_range`. It guarantees that the number of elements in a range can be computed in constant time (either by calling the member `size()` or by computing the difference between the beginning and the end).

If a range type provides `size()` (as a member function or as a free-standing function), this concept is fulfilled by default. To opt out from this concept (e.g., because it iterates over all elements to yield its result), you can and should set `std::disable_sized_range<`*Rg*`>` to `true`:

```
class MyCont {
  ...
  std::size_t size() const;    // assume this is expensive, so that this is not a sized range
};
// opt out from concept std::ranges::sized_range:
constexpr bool std::ranges::disable_sized_range<MyCont> = true;
```

`std::ranges::range` vs. `std::ranges::view`

A similar example of a semantic constraint is the concept `std::ranges::view`. Besides some syntactic constraints, it also guarantees that move constructor/assignment, copy constructor/assignment (if available), and destructor have constant complexity (the time they take does not depend on the number of elements).

An implementor can provide the corresponding guarantee by deriving publicly from either `std::ranges::view_base` or `std::ranges::view_interface<>` or by setting the template specialization `std::ranges::enable_view<`*Rg*`>` to `true`.

`std::invocable` vs. `std::regular_invocable`

A simple example of a semantic constraint is the difference between the concepts `std::invocable` and `std::regular_invocable`. The latter guarantees not to modify the state of the passed operation and the passed arguments.

However, we cannot check the difference between these two concepts with a compiler. Therefore, the concept `std::regular_invocable` documents the ***intention*** of the specified API. Often, for simplicity, just `std::invocable` is used.

`std::weakly_incrementable` vs. `std::incrementable`

Besides certain syntactic differences, there are also semantic differences between the concepts `incrementable` and `weakly_incrementable`:

- `incrementable` requires that each increment of the same value gives the same result.
- `weakly_incrementable` requires only that a type supports the increment operators. Incrementing the same value may yield different results.

Therefore:

- When `incrementable` is satisfied, you can iterate multiple times from a starting value over a range.
- When only `weakly_incrementable` is satisfied, you can iterate over a range only once. A second iteration with the same starting value might yield different results.

This difference matters for iterators: input stream iterators (iterators that read values from a stream) can iterate only once because the next iteration yields different values. Consequently, input stream iterators satisfy the `weakly_incrementable` concept but not the `incrementable` concept. However, the concepts cannot be used to check for this difference:

```
std::weakly_incrementable<std::istream_iterator<int>>   // yields true
std::incrementable<std::istream_iterator<int>>          // OOPS: also yields true
```

The reason is that the difference is a semantic constraint that cannot be checked at compile time. Therefore, the concepts can be used to *document* the constraints:

```
template<std::weakly_incrementable T>
void algo1(T beg, T end);                   // single-pass algorithm
```

```
template<std::incrementable T>
void algo2(T beg, T end);                    // multi-pass algorithm
```

Note that we use different names for the algorithms here. Due to the fact that we cannot check the semantic difference of the constraints, it is up to the programmer to not pass an input stream iterator:

```
algo1(std::istream_iterator<int>{std::cin},  // OK
      std::istream_iterator<int>{});

algo2(std::istream_iterator<int>{std::cin},  // OOPS: violates constraint
      std::istream_iterator<int>{});
```

However, you cannot distinguish between two implementations based on this difference:

```
template<std::weakly_incrementable T>
void algo(T beg, T end);                     // single-pass implementation

template<std::incrementable T>
void algo(T beg, T end);                     // multi-pass implementation
```

If you pass an input stream iterator here, the compiler will incorrectly use the multi-pass implementation:

```
algo(std::istream_iterator<int>{std::cin},   // OOPS: calls the wrong overload
     std::istream_iterator<int>{});
```

Fortunately, there is a solution here, because for this semantic difference, C++98 had already introduced iterator traits, which are used by the iterator concepts. If you use these concepts (or the corresponding range concepts), everything works fine:

```
template<std::input_iterator T>
void algo(T beg, T end);                     // single-pass implementation

template<std::forward_iterator T>
void algo(T beg, T end);                     // multi-pass implementation

algo(std::istream_iterator<int>{std::cin},   // OK: calls the right overload
     std::istream_iterator<int>{});
```

You should prefer the more specific concepts for iterators and ranges, which also honor the new and slightly modified iterator categories.

3.5 Design Guidelines for Concepts

Let us look at some guidelines for how to use concepts. Note that concepts are new and we are still learning how to use them best. In addition, improved support for concepts might change some guidelines over time.

3.5.1 Concepts Should Group Requirements

Introducing a concept for each attribute or functionality of types is certainly too fine-grained. As a consequence, we would get far too many concepts that a compiler has to deal with and that we would all have to specify as constraints.

Therefore, concepts should provide common and typical aspects that separate different categories of requirements or types. However, there are corner cases.

The C++ standard library provides a good example of a design that might follow this approach. Most concepts are provided to categorize types such as ranges, iterators, functions, and so on as a whole. However, to support subsumption and ensure that concepts are consistent, several basic concepts (such as std::movable) are provided.

The consequence is a pretty complex subsumption graph. The chapter that describes the C++ standard concepts groups the concepts accordingly.

3.5.2 Define Concepts with Care

Concepts subsume, which means that a concept can be a subset of another concept, so that in overload resolution the more constrained concept is preferred.

However, requirements and constraints can be defined in various ways. For a compiler, it might not be easy to find out whether a set of requirements is a subset of another set of requirements.

For example, when a concept for two template parameters is commutative (so that the order of two parameters should not matter), a concept needs careful design. For details and an example, see the discussion of how the concept `std::same_as` is defined.

3.5.3 Concepts versus Type Traits and Boolean Expressions

Concepts are more than just expressions that evaluate Boolean results at compile time. You should usually prefer them over type traits and other compile-time expressions.

However, concepts have a couple of benefits:

- They subsume.
- They can be used as type constraints directly in front of template parameters or `auto`.
- They can be used with the compile-time `if` (`if constexpr`) as introduced before.

Benefit From Subsumption

The main benefit of concepts is that they subsume. Type traits do not subsume.

Consider the following example, where we overload a function `foo()` with two requirements defined as **type traits**:

```
template<typename T, typename U>
requires std::is_same_v<T, U>              // using traits
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::is_same_v<T, U> && std::is_integral_v<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // ERROR: ambiguity: both requirements are true
```

The problem is that if both requirements evaluate to `true`, both overloads fit and there is no rule that one of them has priority over the other. Therefore, the compiler stops compiling with an ambiguity error.

If we use the corresponding **concepts** instead, the compiler finds out that the second requirement is a specialization and prefers it if both requirements are met:

```
template<typename T, typename U>
requires std::same_as<T, U>                // using concepts
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<T, U> && std::integral<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

Benefit From Using Concepts with `if constexpr`

C++17 introduced a compile-time if that allows us to switch between code depending on certain compile-time conditions.

For example (as introduced before):

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)      // for floating-point value types
{
  if constexpr(std::is_floating_point_v<T>) {
    ...   // special code for floating-point values
  }
  coll.push_back(val);
}
```

When generic code must provide different implementations for different kinds of arguments, but the signature is the same, using this approach can be way more readable than providing overloaded or specialized templates.

However, you cannot use `if constexpr` to provide different APIs, to allow others to add other overloads or specializations later on, or to disable this template in some cases completely. However, remember that you can constrain member functions to enable or disable parts of an API based on requirements.

3.6 Afternotes

Since C++98, C++ language designers have been exploring how to constrain the parameters of templates with concepts. There have been multiple approaches for introducing concepts in the C++ programming language (e.g., see `http://wg21.link/n1510` by Bjarne Stroustrup). However, the C++ standards committee has not been able to agree on an appropriate mechanism before C++20.

For the C++11 working draft, there was even a very rich concept approach adopted, which was later dropped because it turned out to be too complex. After that, based on `http://wg21.link/n3351` a new approach called ***Concepts Lite*** was proposed by Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis in `http://wg21.link/n3580`. As a consequence, a ***Concepts T****echnical* ***S****pecification* was opened starting with `http://wg21.link/n4549`.

Over time, various improvement were made especially according to the experience of implementing the ranges library.

`http://wg21.link/p0724r0` proposed to apply the *Concepts TS* to the C++20 working draft. The finally accepted wording was formulated by Andrew Sutton in `http://wg21.link/p0734r0`.

Various fixes and improvement were proposed and accepted afterwards. The most visible one was the switch to "standard case" (only lowercase letters and underscores) for the names of the standard concepts as proposed in `http://wg21.link/p1754r1`.

Chapter 4

Concepts, Requirements, and Constraints in Detail

This chapter discusses several details of concepts, requirements, and constraints.

In addition, the following chapter lists and discusses all standard concepts that the C++20 standard library provides.

4.1 Constraints

To specify requirements for generic parameters you need ***constraints***, which are used at compile time to decide whether to instantiate and compile a template or not.

You can constrain function templates, class templates, variable templates, and alias templates.

An ordinary *constraint* is usually specified by using a ***requires clause***. For example:

```
template<typename T>
void foo(const T& arg)
requires MyConcept<T>
...
```

In front of a template parameter or `auto` you can also use a concept as a ***type constraint*** directly:

```
template<MyConcept T>
void foo(const T& arg)
...
```

Alternatively:

```
void foo(const MyConcept auto& arg)
...
```

4.2 `requires` Clauses

A ***requires clause*** uses the keyword **`requires`** with a compile-time Boolean expression to restrict the availability of the template. The Boolean expression can be:

- An ad-hoc Boolean compile-time expression
- A *concept*
- A *requires expression*

All constraints can also be used wherever a Boolean expression can be used (especially as an `if constexpr` condition).

4.2.1 Using `&&` and `||` in `requires` Clauses

To combine multiple constraints in requires clauses, we can use the operator `&&`. For example:

```
template<typename T>
requires (sizeof(T) > 4)                            // ad-hoc Boolean expression
         && requires { typename T::value_type; }    // requires expression
         && std::input_iterator<T>                  // concept
void foo(T x) {
  ...
}
```

The order of the constraints does not matter.

We can also express “alternative” constraints using the operator `||`. For example:

```
template<typename T>
requires std::integral<T> || std::floating_point<T>
T power(T b, T p);
```

Specifying alternative constraints is rarely necessary and should not be done casually because excessive use of the operator `||` in requires clauses may potentially tax compilation resources (i.e., make compilation noticeably slower).

A single constraint can also involve multiple template parameters. That way, constraints can impose a relationship between multiple types (or values). For example:

```
template<typename T, typename U>
requires std::convertible_to<T, U>
auto f(T x, U y) {
  ...
}
```

The operators `&&` and `||` are the only operators you can use to combine multiple constraints without having to use parentheses. For everything else, use parentheses (which formally pass an ad-hoc Boolean expression to the requires clause).

4.3 Ad-hoc Boolean Expressions

The basic way to formulate constraints for templates is to use a requires clause: `requires` followed by a Boolean expression. After `requires`, the constraint can use any compile-time Boolean expressions, not just concepts or requires expressions. These expressions may especially use:

- Type predicates, such as type traits
- Compile-time variables (defined with `constexpr` or `constinit`)
- Compile-time functions (defined with `constexpr` or `consteval`)

Let us look at some examples of using ad-hoc Boolean expressions to restrict the availability of a template:

- Available only if `int` and `long` have a different size:

  ```
  template<typename T>
  requires (sizeof(int) != sizeof(long))
  ...
  ```

- Available only if `sizeof(T)` is not too large:

  ```
  template<typename T>
  requires (sizeof(T) <= 64)
  ...
  ```

- Available only if the non-type template parameter `Sz` is greater than zero:

  ```
  template<typename T, std::size_t Sz>
  requires (Sz > 0)
  ...
  ```

- Available only for raw pointers and the `nullptr`:

  ```
  template<typename T>
  requires (std::is_pointer_v<T> || std::same_as<T, std::nullptr_t>)
  ...
  ```

  `std::same_as` is a new standard concept. Instead, you could also use the standard type trait `std::is_same_v<>`:

  ```
  template<typename T>
  requires (std::is_pointer_v<T> || std::is_same_v<T, std::nullptr_t>)
  ...
  ```

- Available only if the argument cannot be used as a string:

  ```
  template<typename T>
  requires (!std::convertible_to<T, std::string>)
  ...
  ```

  `std::convertible_to` is a new standard concept. You could also use the standard type trait `std::is_convertible_v<>`:

  ```
  template<typename T>
  requires (!std::is_convertible_v<T, std::string>)
  ...
  ```

- Available only if the argument is a pointer (or pointer-like object) to an integral value:

  ```
  template<typename T>
  requires std::integral<std::remove_reference_t<decltype(*std::declval<T>())>>
  ...
  ```

  Note that `operator*` usually yields a reference that is not an integral type. Therefore, we do the following:
  - Assume we have an object of type `T`: `std::declval<T>()`
  - Call `operator*` for the object: `*`
  - Ask for its type: `decltype()`
  - Remove referenceness: `std::remove_reference_v<>`
  - Check whether it is an integral type: `std::integral<>`

  This constraint would also be satisfied by a `std::optional<int>`.

  `std::integral` is a new standard concept. You could also use the standard type trait `std::is_integral_v<>`.
- Available only if the non-type template parameters `Min` and `Max` have a greatest common divisor (GCD) that is greater than one:

  ```
  template<typename T>
  constexpr bool gcd(T a, T b);    // greatest common divisor (forward declaration)

  template<typename T, int Min, int Max>
  requires (gcd(Min, Max) > 1)     // available if there is a GCD greater than 1
  ...
  ```
- Disable a template (temporarily):

  ```
  template<typename T>
  requires false                   // disable the template
  ...
  ```

In non-trivial cases, you need parentheses around the whole requires expression or parts of it. You can only skip parentheses if you use only identifiers, `::`, and `<...>` optionally combined with `&&` and `||`. For example:

```
requires std::convertible_to<T, int>      // no parentheses needed here
         &&
         (!std::convertible_to<int, T>)   // ! forces the need for parentheses
```

4.4 `requires` Expressions

Requires expressions (which are distinct from *requires clauses*) provide a simple but flexible syntax for specifying multiple requirements on one or multiple template parameters. You can specify:

- Required type definitions
- Expressions that have to be valid
- Requirements on the types that expressions yield

A *requires expression* starts with requires followed by an optional parameter list and then a block of requirements (all ending with semicolons). For example:

```
template<typename Coll>
... requires {
      typename Coll::value_type::first_type;    // elements/values have first_type
      typename Coll::value_type::second_type;   // elements/values have second_type
    }
```

The optional parameter list allows you to introduce a set of “dummy variables” that can be used to express requirements in the body of the requires expression:

```
template<typename T>
... requires(T x, T y) {
      x + y;    // supports +
      x - y;    // supports -
    }
```

These parameters are never replaced by arguments. Therefore, it usually does not matter whether you declare them by value or by reference.

The parameters also allow us to introduce (parameters of) sub-types:

```
template<typename Coll>
... requires(Coll::value_type v) {
      std::cout << v;     // supports output operator
    }
```

This requirements checks whether Coll::value_type is valid and whether objects of this type support the output operator.

Note that type members in this parameter list do not have to be qualified with typename.

When using this to check only whether Coll::value_type is valid, you do not need anything in the body of the block of the requirements. However, the block cannot be empty. Therefore, you could simply use true in that case:

```
template<typename Coll>
... requires(Coll::value_type v) {
      true;               // dummy requirement because the block cannot be empty
    }
```

4.4.1 Simple Requirements

Simple requirements are just expressions that have to be well-formed. This means that the calls have to compile. The calls are not performed, meaning that it does not matter what the operations yield.

For example:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p;                // operator* has to be supported for T2
      p[0];              // operator[] has to be supported for int as index
      p->value();       // calling a member function value() without arguments has to be possible
      *p > val;         // support the comparison of the result of operator* with T1
      p == nullptr;     // support the comparison of a T2 with a nullptr
    }
```

The last call does not require that `p` ***is*** the `nullptr` (to require that, you have to check whether `T2` is type `std::nullptr_t`). Instead, we require that we ***can*** compare an object of type `T2` with `nullptr`.

It usually does not make sense to use the operator `||`. A simple requirement such as

```
*p > val || p == nullptr;
```

does ***not*** require that either the left or the right sub-expression is possible. It formulates the requirement that we can combine the results of both sub-expressions with the operator `||`.

To require either one of the two sub-expressions, you have to use:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p > val;           // support the comparison of the result of operator* with T1
    }
    || requires(T2 p) {   // OR
      p == nullptr;       // support the comparison of a T2 with nullptr
    }
```

Note also that this concept does ***not*** require that `T` is an integral type:

```
template<typename T>
... requires {
      std::integral<T>;           // OOPS: does not require T to be integral
      ...
    };
```

The concept requires only that the expression `std::integral<T>` is valid, which is the case for all types. The requirements that `T` is integral you have to formulate as follows:

```
template<typename T>
... std::integral<T> &&           // OK, does require T to be integral
    requires {
      ...
    };
```

Alternatively, as follows:

```
template<typename T>
... requires {
      requires std::integral<T>;  // OK, does require T to be integral
      ...
    };
```

4.4.2 Type Requirements

Type requirements are expressions that have to be well-formed when using a name of a type. This means that the specified name has to be defined as a valid type.

For example:

```
template<typename T1, typename T2>
... requires {
  typename T1::value_type;                 // type member value_type required for T1
  typename std::ranges::iterator_t<T1>;    // iterator type required for T1
  typename std::common_type_t<T1, T2>;     // T1 and T2 have to have a common type
}
```

For all type requirements, if the type exists but is `void`, then the requirement is met.

Note that you can only check for names given to types (names of classes, enumeration types, from `typedef` or `using`). You cannot check for other type declarations using the type:

```
template<typename T>
... requires {
  typename int;               // ERROR: invalid type requirement
  typename T&;                // ERROR: invalid type requirement
}
```

The way to test the latter is to declare a corresponding parameter:

```
template<typename T>
... requires(T&) {
  true;       // some dummy requirement
};
```

Again, the requirement checks whether using the passed type(s) to define another type is valid. For example:

```
template<std::integral T>
class MyType1 {
  ...
};

template<typename T>
requires requires {
  typename MyType1<T>;        // instantiation of MyType1 for T would be valid
}
void mytype1(T) {
  ...
}

mytype1(42);     // OK
mytype1(7.7);    // ERROR
```

Therefore, the following requirement does ***not*** check whether there is a standard hash function for type T:

```
template<typename T>
concept StdHash = requires {
  typename std::hash<T>;    // does not check whether std::hash<> is defined for T
};
```

The way to require a standard hash function is to try to create or use it:

```
template<typename T>
concept StdHash = requires {
  std::hash<T>{};           // OK, checks whether we can create a standard hash function for T
};
```

Note that simple requirements check only whether a requirement is **valid**, not whether it is fulfilled. For this reason:

- It does not make sense to use type functions that always yield a value:

  ```
  template<typename T>
  ... requires {
    std::is_const_v<T>;                 // not useful: always valid (doesn't matter what it yields)
  }
  ```

  To check for constness, use:

  ```
  template<typename T>
  ... std::is_const_v<T>   // ensure that T is const
  ```

  Inside a requires expression, you can use a nested requirement (see below).

- It does not make sense to use type functions that always yield a type:

  ```
  template<typename T>
  ... requires {
    typename std::remove_const_t<T>;    // not useful: always valid (yields a type)
  }
  ```

  The requirement checks only whether the type expression yields a type, which is always the case.

It also does not make sense to use type functions that may have undefined behavior. For example, the type trait std::make_unsigned<> requires that the passed argument is an integral type other than bool. If the passed type is not an integral type, you get undefined behavior. Therefore, you should ***not*** use std::make_unsigned<> as requirement without constraining the type you call it for:

```
template<typename T>
... requires {
  std::make_unsigned<T>::type;   // not useful as type requirement (valid or undefined behavior)
}
```

In this case, the requirement can only be fulfilled or results in undefined behavior (which might mean that the requirement is still fulfilled). Instead, you should also constrain the type T for which you can use a nested requirement:

```
template<typename T>
... requires {
  requires (std::integral<T> && !std::same_as<T, bool>);
  std::make_unsigned<T>::type;     // OK
}
```

4.4.3 Compound Requirements

Compound requirements allow us to combine the abilities of simple and type requirements. In this case, you can specify an expression (inside a block of braces) and then add one or both of the following:

- noexcept to require that the expression guarantees not to throw
- -> *type-constraint* to apply a concept on what the expression evaluates to

Here are some examples:

```
template<typename T>
... requires(T x) {
  { &x } -> std::input_or_output_iterator;
  { x == x };
  { x == x } -> std::convertible_to<bool>;
  { x == x }noexcept;
  { x == x }noexcept -> std::convertible_to<bool>;
}
```

Note that the type constraint after the -> takes the resulting type as its first template argument. That means:

- In the first requirement, we require that the concept std::input_or_output_iterator is satisfied when using operator& for an object of type T (std::input_or_output_iterator<decltype(&x)> yields true).

  You could also specify this as follows:

  ```
  { &x } -> std::is_pointer_v<>;
  ```

- In the last requirement, we require that we can use the result of operator== for two objects of type T as bool (the concept std::convertible_to is satisfied when passing the result of operator== for two objects of type T and bool as arguments).

Requires expressions can also express the need for associated types. For example:

```
template<typename T>
... requires(T coll) {
  { *coll.begin() } -> std::convertible_to<T::value_type>;
}
```

However, you cannot specify type requirements using nested types. For example, you cannot use them to require that the return value of calling the operator * yields an integral value. The problem is that the return value is a reference that you have to dereference first:

```
std::integral<std::remove_reference_t<T>>
```

and you cannot use such a nested expression with a type trait in a result of a requires expression:

```
template<typename T>
concept Check = requires(T p) {
  { *p } -> std::integral<std::remove_reference_t<>>;   // ERROR
  { *p } -> std::integral<std::remove_reference_t>;     // ERROR
};
```

You either have to define a corresponding concept first:

```
template<typename T>
concept UnrefIntegral = std::integral<std::remove_reference_t<T>>;

template<typename T>
concept Check = requires(T p) {
  { *p } -> UnrefIntegral;   // OK
};
```

Alternatively, you have to use a nested requirement.

4.4.4 Nested Requirements

Nested requirements can be used to specify additional constraints inside a requires expression. They start with `requires` followed by a compile-time Boolean expression, which might itself again be or use a requires expression. The benefit of nested requirements is that we can ensure that a compile-time expression (that uses parameters or sub-expressions of the requires expression) yields a certain result instead of ensuring only that the expression is valid.

For example, consider a concept that has to ensure that both the operator `*` and the operator `[]` yield the same type for a given type. By using nested requirements, we can specify this as follows:

```
template<typename T>
concept DerefAndIndexMatch = requires (T p) {
                               requires std::same_as<decltype(*p),
                                                     decltype(p[0])>;
                             };
```

The good thing is that we have an easy syntax here for "*assume we have an object of type* `T`." We do not have to use a requires expression here; however, the code then has to use `std::declval<>()`:

```
template<typename T>
concept DerefAndIndexMatch = std::same_as<decltype(*std::declval<T>()),
                                          decltype(std::declval<T>()[0])>;
```

As another example, we can use a nested requirement to solve the problem just introduced to specify a complex type requirement on an expression:[1]

```
template<typename T>
concept Check = requires(T p) {
  requires std::integral<std::remove_cvref_t<decltype(*p)>>;
};
```

Note the following difference inside a requires expression:

```
template<typename T>
... requires {
  !std::is_const_v<T>;            // OOPS: checks whether we can call is_const_v<>
  requires !std::is_const_v<T>;   // OK: checks whether T is not const
}
```

[1] Thanks to Hannes Hauswedell for pointing this out.

Here, we use the type trait `is_const_v<>` without and with `requires`. However, only the second requirement is useful:

- The first expression requires only that ***checking*** for constness ***and negating*** the result is valid. This requirement is always met (even if `T` is `const int`) because doing this check is always valid. This requirement is worthless.
- The second expression with `requires` has to be ***fulfilled***. The requirement is met if `T` is `int` but not if `T` is `const int`.

4.5 Concepts in Detail

By defining a `concept`, you can introduce a name for one or more *constraints*.

Templates (function, class, variable, and alias templates) can use concepts to constrain their ability (via a requires clause or as a direct type constraint for a template parameter). However, concepts are also Boolean compile-time expressions (type predicates) that you can use wherever you have to check something for a type (such as in an `if constexpr` condition).

4.5.1 Defining Concepts

Concepts are defined as follows:

```
template<...>
concept name = ... ;
```

The equal sign is required (you cannot declare a concept without defining it and you cannot use braces here). After the equal sign, you can specify any compile-time expression that converts to `true` or `false`.

Concepts are much like `constexpr` variable templates of type `bool`, but the type is not explicitly specified:

```
template<typename T>
concept MyConcept = ... ;

std::is_same<MyConcept<...>, bool>   // yields true
```

This means that both at compile time and at runtime you can always use a concept where the value of a Boolean expression is needed. However, you cannot take the address because there is no object behind it (it is a prvalue).

The template parameters may not have constraints (you cannot use a concept to define a concept).

You cannot define concepts inside a function (as is the case for all templates).

4.5.2 Special Abilities of Concepts

Concepts have special abilities.

Consider, for example, the following concept:

```
template<typename T>
concept IsOrHasThisOrThat = ... ;
```

Compared to a definition of a Boolean variable template (which is the usual way type traits are defined):

```
template<typename T>
inline constexpr bool IsOrHasThisOrThat = ... ;
```

we have the following differences:

- Concepts do not represent code. They have no type, storage, lifetime, or any other properties associated with objects.

  By instantiating them at compile time for specific template parameters, their instantiation just becomes `true` or `false`. Therefore, you can use them wherever you can use `true` or `false` and you get all properties of these literals.
- Concepts do not have to be declared as `inline`. They implicitly are `inline`.
- Concepts can be used as type constraints:

  ```
  template<IsOrHasThisOrThat T>
  ...
  ```

  Variable templates cannot be used that way.
- Concepts are the only way to give *constraints* a name, which means that you need them to decide whether a constraint is a special case of another constraint.
- Concepts subsume. To let the compiler decide whether a constraint implies another constraint (and is therefore special), the constraints have to be formulated as concepts.

4.5.3 Concepts for Non-Type Template Parameters

Concepts can also be applied to non-type template parameters (NTTP). For example:

```
template<auto Val>
concept LessThan10 = Val < 10;

template<int Val>
requires LessThan10<Val>
class MyType {
  ...
};
```

As a more useful example, we can use a concept to constrain the value of a non-type template parameter to be a power of two:

lang/conceptnttp.cpp

```
#include <bit>

template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));

template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
```

```
  ...
};

int main()
{
   Memory<int, 8> m1;      // OK
   Memory<int, 9> m2;      // ERROR
   Memory<int, 32> m3;     // OK
   Memory<int, true> m4;   // OK
   ...
}
```

The concept `PowerOf2` takes a value instead of a type as a template parameter (here, using `auto` to not require a specific type):

```
template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));
```

The concept is satisfied when the new standard function `std::has_single_bit()` yields `true` for the passed value (having only one bit set means that a value is a power of two). Note that `std::has_single_bit()` requires that we have an unsigned integral value. By casting to `unsigned`, programmers can pass signed integral values and reject types that cannot be converted to an unsigned integral value.

The concept is then used to require that a class `Memory`, taking a type and a size, accepts only sizes that are a power of two:

```
template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
  ...
};
```

Note that you cannot write the following:

```
template<typename T, PowerOf2 auto Val>
class Memory {
  ...
};
```

This puts the requirement on the ***type*** of `Val`; however, the concept `PowerOf2` does not constrain a type; it constrains the value.

4.6 Using Concepts as Type Constraints

As introduced, concepts can be used as type constraints. There are different places where type constraints can be used:

- In the declaration of a template type parameter
- In the declaration of a call parameter declared with `auto`
- As a requirement in a compound requirement

For example:

```
template<std::integral T>                       // type constraint for a template parameter
class MyClass {
  ...
};

auto myFunc(const std::integral auto& val) {    // type constraint for an auto parameter
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::integral;                 // type constraint for return type
  };
```

Here, we use unary constraints that are called for a single parameter or type returned by an expression.

Type Constraints with Multiple Parameters

You can also use constraints with multiple parameters, for which the parameter type or return value is then used as the first argument:

```
template<std::convertible_to<int> T>                     // conversion to int required
class MyClass {
  ...
};

auto myFunc(const std::convertible_to<int> auto& val) {  // conversion to int required
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::convertible_to<int>;               // conversion to int required
  };
```

Another example often used is to constrain the type of a callable (function, function object, lambda) to require that you can pass a certain number of arguments of certain types using the concepts std::invocable or std::regular_invocable: for example, to require the passing of an operation that takes an `int` and a `std::string`, you have to declare:

```
template<std::invocable<int, std::string> Callable>
void call(Callable op);
```

or:

```
void call(std::invocable<int, std::string> auto op);
```

The difference between `std::invocable` and `std::regular_invocable` is that the latter guarantees not to modify the passed operation and arguments. That is a semantic difference that helps only to document the intention. Often, just `std::invocable` is used.

Type Constraints and `auto`

Type constraints can be used in all places where `auto` can be used. The major application of this feature is to use the type constraints for the function parameters declared with `auto`. For example:

```
void foo(const std::integral auto& val)
{
  ...
}
```

However, you can also use type constraint for `auto` as follows:

- To constrain declarations:

  ```
  std::floating_point auto val1 = f();            // valid if f() yields floating-point value

  for (const std::integral auto& elem : coll) {   // valid if elements are integral values
    ...
  }
  ```

- To constrain return types:

  ```
  std::copyable auto foo(auto) {                  // valid if foo() returns copyable value
    ...
  }
  ```

- To constrain non-type template parameters:

  ```
  template<typename T, std::integral auto Max>
  class SizedColl {
    ...
  };
  ```

This also works with concepts that take multiple parameters:

```
template<typename T, std::convertible_to<T> auto DefaultValue>
class MyType {
  ...
};
```

For another example, see the support for lambdas as non-type template parameters.

4.7 Subsuming Constraints with Concepts

Two concepts can have a subsuming relationship. That is, one concept can be specified such that it restricts one or more other concepts. The benefit is that overload resolution then prefers the more constrained generic code over the less constrained generic code when both constraints are satisfied.

For example, assume that we introduce the following two concepts:

```
template<typename T>
concept GeoObject = requires(T obj) {
    { obj.width() } -> std::integral;
    { obj.height() } -> std::integral;
    obj.draw();
  };

template<typename T>
concept ColoredGeoObject =
  GeoObject<T> &&               // subsumes concept GeoObject
  requires(T obj) {             // additional constraints
    obj.setColor(Color{});
    { obj.getColor() } -> std::convertible_to<Color>;
  };
```

The concept `ColoredGeoObject` explicitly *subsumes* the concept `GeoObject` because it explicitly formulates the constraint that type `T` also has to satisfy the concept `GeoObject`.

As a consequence, we do not get an ambiguity error, when we overload templates for both concepts and both concepts are satisfied, Overload resolution prefers the concept that subsumes the other(s):

```
template<GeoObject T>
void process(T)          // called for objects that do not provide setColor() and getColor()
{
  ...
}

template<ColoredGeoObject T>
void process(T)          // called for objects that provide setColor() and getColor()
{
  ...
}
```

Constraint subsumption works only when concepts are used. There is no automatic subsumption when one concept/constraint is more special than another.

Constraints and concepts do *not* subsume based only on requirements. Consider the following example:[2]

[2] This was in fact the example discussed regarding this feature during standardization. Thanks to Ville Voutilainen for pointing this out.

```
// declared in a header file:
template<typename T>
concept GeoObject = requires(T obj) {
                      obj.draw();
                    };

// declared in another header file:
template<typename T>
concept Cowboy = requires(T obj) {
                   obj.draw();
                   obj = obj;
                 };
```

Assume that we overload a function template for both `GeoObject` and `Cowboy`:

```
template<GeoObject T>
void print(T) {
  ...
}

template<Cowboy T>
void print(T) {
  ...
}
```

We do not want that for a `Circle` or `Rectangle`, which both have a `draw()` member function, the call to `print()` prefers the `print()` for cowboys just because the `Cowboy` concept is more special. We want to see that there are two possible `print()` functions that in this case collide.

The effort to check for subsumptions is evaluated only for concepts. Overloading with different constraints is ambiguous if no concepts are used:

```
template<typename T>
requires std::is_convertible_v<T, int>
void print(T) {
  ...
}

template<typename T>
requires (std::is_convertible_v<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);    // ERROR: ambiguous (if both constraints are true)
```

When using concepts instead, this code works:

```
template<typename T>
requires std::convertible_to<T, int>
```

```
void print(T) {
  ...
}

template<typename T>
requires (std::convertible_to<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);    // OK
```

One reason for this behavior is that it takes compile time to process dependencies between concepts in detail.

The concepts provided by the C++ standard library are carefully designed to subsume other concepts when this makes sense. In fact, the standard concepts build a pretty complex subsumption graph. For example:

- `std::random_access_range` subsumes `std::bidirectional_range`, both subsume the concept `std::forward_range`, all three subsume `std::input_range`, and all of them subsume `std::range`. However, `std::sized_range` only subsumes `std::range` and none of the others.
- `std::regular` subsumes `std::semiregular`, while both subsume `std::copyable` and `std::default_initializable` (which subsume several other concepts such as `std::movable`, `std::copy_constructible`, and `std::destructible`).
- `std::sortable` subsumes `std::permutable` and both subsume `std::indirectly_swappable` for both parameters being the same type.

4.7.1 Indirect Subsumptions

Constraints can even subsume indirectly.[3] This means that overload resolution can still prefer one overload or specialization over the other, even though their constraints are not defined in terms of each other.

For example, assume that you have defined the following two concepts:

```
template<typename T>
concept RgSwap = std::ranges::input_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

When we now overload two functions for these two concepts and pass an object that fits both concepts, this is not ambiguous:

```
template<RgSwap T>
void foo1(T) {
  std::cout << "foo1(RgSwap)\n";
}
```

[3] Thanks to Arthur O'Dwyer for pointing this out.

```
template<ContCopy T>
void foo1(T) {
  std::cout << "foo1(ContCopy)\n";
}

foo1(std::vector<int>{});       // OK: both fit, ContCopy is more constrained
```

The reason is that `ContCopy` subsumes `RgSwap` because:

- The concept `contiguous_range` is defined in terms of the concept `input_range`.
  (It implies `random_access_range`, which implies `bidirectional_range`, which implies `forward_range`, which implies `input_range`.)
- The concept `copyable` is defined in terms of the concept `swappable`.
  (It implies `movable`, which implies `swappable`.)

However, with the following declarations, we get an ambiguity when both concepts fit:

```
template<typename T>
concept RgSwap = std::ranges::sized_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

The reason for is that neither of the concepts `contiguous_range` or `sized_range` imply the other.

Also, for the following declarations, no concept subsumes the other:

```
template<typename T>
concept RgCopy = std::ranges::input_range<T> && std::copyable<T>;

template<typename T>
concept ContMove = std::ranges::contiguous_range<T> && std::movable<T>;
```

On the one hand, `ContMove` is more constrained because `contiguous_range` implies `input_range`; however, on the other hand, `RgCopy` is more constrained because `copyable` implies `movable`.

To avoid confusion, do not make too many assumptions about concepts subsuming each other. When in doubt, specify all the concepts you require.

4.7.2 Defining Commutative Concepts

To implement subsumptions correctly, you have to be careful. A good example is the implementation of the concept `std::same_as`, which checks whether two template parameters have the same type.

To understand why defining the concept is not trivial, let us assume that we define our own concept `SameAs` just as follows:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;          // define concept SameAs
```

This definition is good enough for cases like this:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;            // define concept SameAs

template<typename T, typename U>
requires SameAs<T, U>                             // use concept SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<T, U> && std::integral<T>         // use concept SameAs again
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

Here, the second definition of `foo()` subsumes the first one, which means that the second `foo()` is called when two arguments of the same integral type are passed and for both `foo()` the constraints are satisfied.

However, note what happens if we slightly modify the parameters when we call `SameAs<>` in the constraints of the second `foo()`:

```
template<typename T, typename U>
requires SameAs<T, U>                             // use concept SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>         // use concept SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // ERROR: ambiguity: both constraints are satisfied without one subsuming the other
```

The problem is that the compiler cannot detect that `SameAs<>` is commutative. For the compiler, the order of the template parameters matters, and therefore, the first requirement is not necessarily a subset of the second requirement.

To solve this problem, we have to design the concept `SameAs` in a way that the order of the arguments does not matter. This requires a helper concept:

```
template<typename T, typename U>
concept SameAsHelper = std::is_same_v<T, U>;

template<typename T, typename U>
concept SameAs = SameAsHelper<T, U> && SameAsHelper<U, T>;   // make commutative
```

Now, the order of the parameters no longer matters for `IsSame<>`:

```
template<typename T, typename U>
requires SameAs<T, U>                             // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>         // use SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);     // OK: second foo() preferred
```

The compiler can find out that the first building block `SameAs<U,T>` is part of a sub-concept of the definition of `SameAs` so that the other building blocks `SameAs<T,U>` and `std::integral<T>` are an extension. Therefore, the second `foo()` now has preference.

Tricky details like this are designed into the concepts that the C++20 standard library provides (see `std::same_as`). Therefore, you should use them when they fit your needs instead of providing your own concepts.

```
template<typename T, typename U>
requires std::same_as<T, U>                       // standard same_as<> is commutative
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<U, T> && std::integral<T>   // so different order does not matter
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);     // OK: second foo() preferred
```

For your own concepts, take as many uses (and abuses) as possible into account. The more the better. Just like any good piece of software, concepts also require good design and test cases.

Chapter 5

Standard Concepts in Detail

This chapter describes all *concepts* of the C++20 standard library in detail.

5.1 Overview of All Standard Concepts

Table *Basic concepts for types and objects* lists the basic concepts for types and objects in general.

Table *Concepts for ranges, iterators, and algorithms* lists the concepts for ranges, views, iterators, and algorithms.

Table *Auxiliary concepts* lists the concepts that are used mainly as building blocks for other concepts and are usually not used directly by application programmers.

5.1.1 Header Files and Namespaces

Standard concepts are defined in different header files:

- Many basic concepts are defined in the header file `<concepts>`, which is included by `<ranges>` and `<iterator>`.
- Concepts for iterators are defined in the header file `<iterator>`.
- Concepts for ranges are defined in the header file `<ranges>`.
- `three_way_comparable` concepts are defined in `<compare>` (which is included by almost every other header file).
- `uniform_random_bit_generator` is defined in `<random>`.

Almost all concepts are defined in the namespace `std`. The only exceptions to this rule are the ranges concepts, which are defined in the namespace `std::ranges`.

Concept	**Constraint**
`integral`	Integral type
`signed_integral`	Signed integral type
`unsigned_integral`	Unsigned integral type
`floating_point`	Floating-point type
`movable`	Supports move initialization/assignment and swaps
`copyable`	Supports move and copy initialization/assignment and swaps
`semiregular`	Supports default initialization, copies, moves, and swaps
`regular`	Supports default initialization, copies, moves, swaps, and equality comparisons
`same_as`	Same types
`convertible_to`	Type convertible to another type
`derived_from`	Type derived from another type
`constructible_from`	Type constructible from others types
`assignable_from`	Type assignable from another type
`swappable_with`	Type swappable with another type
`common_with`	Two types have a common type
`common_reference_with`	Two types have a common reference type
`equality_comparable`	Type supports checks for equality
`equality_comparable_with`	Can check two types for equality
`totally_ordered`	Types support a strict weak ordering
`totally_ordered_with`	Can check two types for strict weak ordering
`three_way_comparable`	Can apply all comparison operators (including the `operator <=>`)
`three_way_comparable_with`	Can compare two types with all comparison operators (including `<=>`)
`invocable`	Type is a callable for specified arguments
`regular_invocable`	Type is a callable for specified arguments (no modifications)
`predicate`	Type is a predicate (callable that returns a Boolean value)
`relation`	A callable type defines a relationship between two types
`equivalence_relation`	A callable type defines an equality relationship between two types
`strict_weak_order`	A callable type defines an ordering relationship between two types
`uniform_random_bit_generator`	A callable type can be used as a random number generator

Table 5.1. Basic concepts for types and objects

Concept	**Constraint**
`default_initializable`	Type is default initializable
`move_constructible`	Type supports move initializations
`copy_constructible`	Type supports copy initializations
`destructible`	Type is destructible
`swappable`	Type is swappable
`weakly_incrementable`	Type supports the increment operators
`incrementable`	Type supports equality-preserving increment operators

Table 5.2. Auxiliary concepts

Concept	**Constraint**
`range`	Type is a range
`output_range`	Type is a range to write to
`input_range`	Type is a range to read from
`forward_range`	Type is a range to read from multiple times
`bidirectional_range`	Type is a range to read forward and backward from
`random_access_range`	Type is a range that supports jumping around over elements
`contiguous_range`	Type is a range with elements in contiguous memory
`sized_range`	Type is a range with cheap size support
`common_range`	Type is a range with iterators and sentinels that have the same type
`borrowed_range`	Type is an lvalue or a borrowed range
`view`	Type is a view
`viewable_range`	Type is or can be converted to a view
`indirectly_writable`	Type can be used to write to where it refers
`indirectly_readable`	Type can be used to read from where it refers
`indirectly_movable`	Type refers to movable objects
`indirectly_movable_storable`	Type refers to movable objects with support for temporaries
`indirectly_copyable`	Type refers to copyable objects
`indirectly_copyable_storable`	Type refers to copyable objects with support for temporaries
`indirectly_swappable`	Type refers to swappable objects
`indirectly_comparable`	Type refers to comparable objects
`input_output_iterator`	Type is an iterator
`output_iterator`	Type is an output iterator
`input_iterator`	Type is (at least) an input iterator
`forward_iterator`	Type is (at least) a forward iterator
`bidirectional_iterator`	Type is (at least) a bidirectional iterator
`random_access_iterator`	Type is (at least) a random-access iterator
`contiguous_iterator`	Type is an iterator to elements in contiguous memory
`sentinel_for`	Type can be used as a sentinel for an iterator type
`sized_sentinel_for`	Type can be used as a sentinel for an iterator type with cheap computation of distances
`permutable`	Type is (at least) a forward iterator that can reorder elements
`mergeable`	Two types can be used to merge sorted elements into a third type
`sortable`	A type is sortable (according to a comparison and projection)
`indirectly_unary_invocable`	Operation can be called with the value type of an iterator
`indirectly_regular_unary_invocable`	Stateless operation can be called with the value type of an iterator
`indirect_unary_predicate`	Unary predicate can be called with the value type of an iterator
`indirect_binary_predicate`	Binary predicate can be called with the value types of two iterators
`indirect_equivalence_relation`	Predicate can be used to check two values of the passed iterator(s) for equality
`indirect_strict_weak_order`	Predicate can be used to order two values of the passed iterator(s)

Table 5.3. Concepts for ranges, iterators, and algorithms

5.1.2 Standard Concepts Subsume

The concepts provided by the C++ standard library are carefully designed to subsume other concepts when this makes sense. In fact, they build a pretty complex subsumption graph. Figure 5.1 gives an impression of how complex it is.

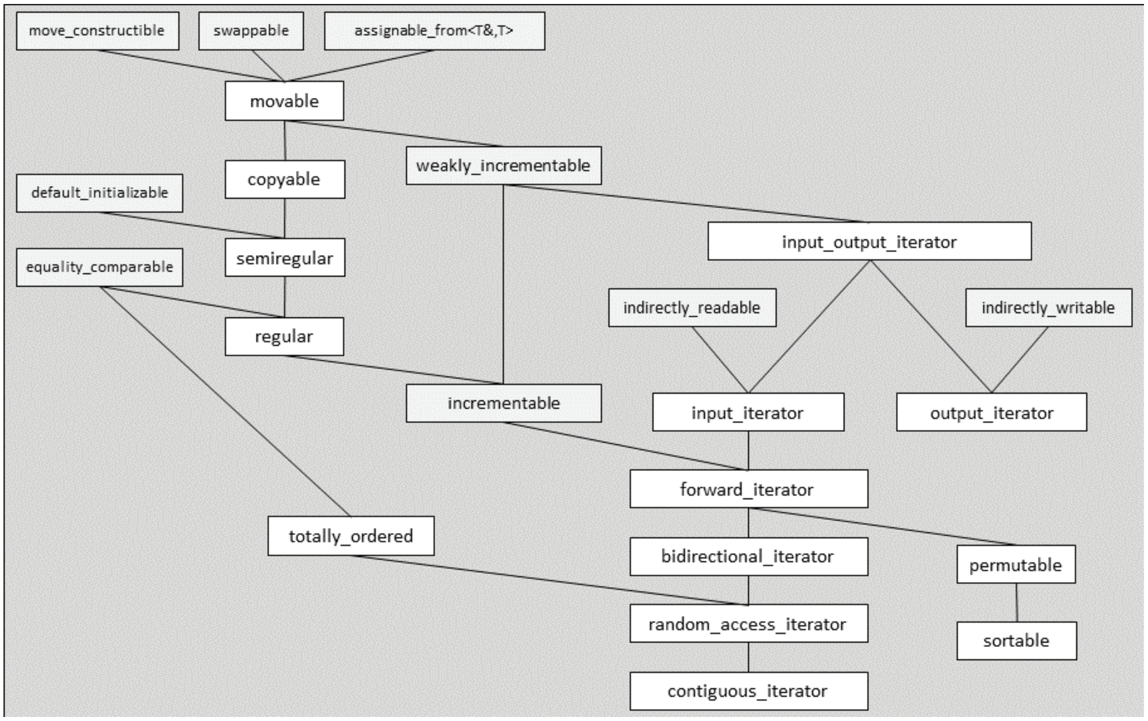


Figure 5.1. Subsumption graph of C++ standard concepts (extract)

For this reason, the description of the concepts lists which other key concepts are subsumed.

5.2 Language-Related Concepts

This section lists the concepts that apply to objects and types in general.

5.2.1 Arithmetic Concepts

`std::`**`integral`**`<`*T*`>`

- Guarantees that type *T* is an integral type (including `bool` and all character types).
- Requires:
  - The type trait `std::is_integral_v<`*T*`>` yields `true`

`std::`**`signed_integral`**`<`*T*`>`

- Guarantees that type *T* is a signed integral type (including signed character types, which `char` might be).
- Requires:
  - `std::integral<`*T*`>` is satisfied
  - The type trait `std::is_signed_v<`*T*`>` yields `true`

`std::`**`unsigned_integral`**`<`*T*`>`

- Guarantees that type *T* is an unsigned integral type (including `bool` and unsigned character types, which `char` might be).
- Requires:
  - `std::integral<`*T*`>` is satisfied
  - `std::signed_integral<`*T*`>` is not satisfied

`std::`**`floating_point`**`<`*T*`>`

- Guarantees that type *T* is a floating-point type (`float`, `double`, or `long double`).
- The concept was introduced to enable the definition of the mathematical constants.
- Requires:
  - The type trait `std::is_floating_point_v<`*T*`>` yields `true`

5.2.2 Object Concepts

For objects (types that are neither references nor functions nor `void`) there is a hierarchy of required basic operations.

`std::`**`movable`**`<`*T*`>`

- Guarantees that type *T* is movable and swappable. That is, you can move construct, move assign, and swap with another object of the type.

- Requires:
  - `std::move_constructible<`*T*`>` is satisfied
  - `std::assignable_from<`*T*`&, `*T*`>` is satisfied
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`copyable`**`<`*T*`>`

- Guarantees that type *T* is copyable (which implies movable and swappable).
- Requires:
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`semiregular`**`<`*T*`>`

- Guarantees that type *T* is semiregular (can default initialize, copy, move, and swap).
- Requires:
  - `std::copyable<`*T*`>` is satisfied
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

`std::`**`regular`**`<`*T*`>`

- Guarantees that type *T* is regular (can default initialize, copy, move, swap, and check for equality).
- Requires:
  - `std::semiregular<`*T*`>` is satisfied
  - `std::equality_comparable<`*T*`>` is satisfied
  - `std::copyable<`*T*`>` is satisfied
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::copy_constructible<`*T*`>` is satisfied
  - `std::assignable_from` for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*`&`
  - `std::swappable<`*T*`>` is satisfied
  - *T* is neither a reference nor a function nor `void`

5.2.3 Concepts for Relationships between Types

`std::`**`same_as`**`<`*T1*`, `*T2*`>`

- Guarantees that the types *T1* and *T2* are the same.
- The concept calls the `std::is_same_v` type trait twice to ensure that the order of the parameters does not matter.
- Requires:
  - The type trait `std::is_same_v<`*T1*`, `*T2*`>` yields `true`
  - The order of *T1* and *T2* does not matter

`std::`**`convertible_to`**`<`*From*`, `*To*`>`

- Guarantees that objects of type *From* are both implicitly and explicitly convertible to objects of type *To*.
- Requires:
  - The type trait `std::is_convertible_v<`*From*`, `*To*`>` yields `true`
  - A `static_cast` from *From* to *To* is supported
  - You can return an object of type *From* as a *To*

`std::`**`derived_from`**`<`*D*`, `*B*`>`

- Guarantees that type *D* is publicly derived from type *B* (or *D* and *B* are the same) so that any pointer of type *D* can be converted to a pointer of type *B*. In other words: the concept guarantees that a *D* reference/pointer can be used as a *B* reference/pointer.
- Requires:
  - The type trait `std::is_base_of_v<`*B*`, `*D*`>` yields `true`
  - The type trait `std::is_convertible_v` for a `const` pointer of type *D* to *B* yields `true`

`std::`**`constructible_from`**`<`*T*`, `*Args...*`>`

- Guarantees that you can initialize an object of type *T* with parameters of types *Args...*
- Requires:
  - `std::destructible<`*T*`>` is satisfied
  - The type trait `std::is_constructible_v<`*T*`, `*Args...*`>` yields true

`std::`**`assignable_from`**`<`*To*`, `*From*`>`

- Guarantees that you can move or copy assign a value of type *From* to a value of type *To*.
  The assignment also has to yield the original *To* object.
- Requires:
  - *To* has to be an lvalue reference
  - `std::common_reference_with<`*To*`, `*From*`>` is satisfied for `const` lvalue references of the types
  - The operator = has to be supported and yield the same type as *To*

`std::`**`swappable_with`**`<`*T1*`,` *T2*`>`

- Guarantees that values of types *T1* and *T2* can be swapped.
- Requires:
  - `std::common_reference_with<`*T1*`,` *T2*`>` is satisfied
  - Any two objects of types *T1* and *T2* can swap values with each other by using `std::ranges::swap()`

`std::`**`common_with`**`<`*T1*`,` *T2*`>`

- Guarantees that types *T1* and *T2* share a common type to which they can be explicitly converted.
- Requires:
  - The type trait `std::common_type_t<`*T1*`,` *T2*`>` yields a type
  - A `static_cast` is supported to their common type
  - References of both types share a `common_reference` type
  - The order of *T1* and *T2* does not matter

`std::`**`common_reference_with`**`<`*T1*`,` *T2*`>`

- Guarantees that types *T1* and *T2* share a `common_reference` type to which they can be explicitly and implicitly converted.
- Requires:
  - The type trait `std::common_reference_t<`*T1*`,` *T2*`>` yields a type
  - Both types are `std::convertible_to` their common reference type
  - The order of *T1* and *T2* does not matter

5.2.4 Comparison Concepts

`std::`**`equality_comparable`**`<`*T*`>`

- Guarantees that objects of type *T* are comparable with the operators `==` and `!=`. The order should not matter.
- The operator `==` for *T* should be symmetric and transitive:
  - `t1==t2` is `true` if and only if `t2==t1`
  - If `t1==t2` and `t2==t3` are `true`, then `t1==t3` is `true`

  However, this a semantic constraint that cannot be checked at compile time.
- Requires:
  - Both `==` and `!=` are supported and yield a value convertible to `bool`

`std::`**`equality_comparable_with`**`<`*T1*`,` *T2*`>`

- Guarantees that objects of types *T1* and *T2* are comparable using the operators `==` and `!=`.
- Requires:
  - `==` and `!=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`

`std::`**`totally_ordered`**`<`*T*`>`

- Guarantees that objects of type *T* are comparable with the operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either equal to, or less than, or greater than each other.
- The concept does ***not*** claim that type `T` has a total order for all values. In fact, the expression `std::totally_ordered<double>` yields `true` even though floating-point values do not have a total order:

  ```
  std::totally_ordered<double>                    // true
  std::totally_ordered<std::pair<double, double>> // true
  std::totally_ordered<std::complex<int>>         // false
  ```

  This concept is therefore provided to check the formal requirements to be able to sort elements. It is used by `std::ranges::less`, which is the default sorting criterion for sorting algorithms. That way, sorting algorithms do not fail to compile if types do not have a total order for all values. Supporting all six basic comparison operators is enough. The values to be sorted should have a total or weak order. However, this a semantic constraint that cannot be checked at compile time.
- Requires:
  - `std::equality_comparable<`*T*`>` is satisfied
  - All comparisons with the operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`

`std::`**`totally_ordered_with`**`<`*T1*`,` *T2*`>`

- Guarantees that objects of types *T1* and *T2* are comparable with the operators `==`, `!=`, `<`, `<=`, `>`, and `>=` so that two values are always either equal to, or less than, or greater than each other.
- Requires:
  - `==`, `!=`, `<`, `<=`, `>`, and `>=` are supported for all comparisons where objects of *T1* and/or *T2* are involved and yield a value of the same type convertible to `bool`

`std::`**`three_way_comparable`**`<`*T*`>`
`std::`**`three_way_comparable`**`<`*T*`,` *Cat*`>`

- Guarantees that objects of type *T* are comparable with the operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and the operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed, `std::partial_ordering` is required.
- This concept is defined in the header file `<compare>`.
- Note that this concept does not imply and subsume `std::equality_comparable` because the latter requires that the operator `==` yields true only for two objects that are equal. With a weak or partial order, this might not be the case.
- Note that this concept does not imply and subsume std::totally_ordered because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - All comparisons with the operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`
  - Any comparison with the operator `<=>` yields a comparison category (which is at least *Cat*)

`std::`**`three_way_comparable_with`**`<`*T1*`, `*T2*`>`
`std::`**`three_way_comparable_with`**`<`*T1*`, `*T2*`, `*Cat*`>`

- Guarantees that any two objects of types *T1* and *T2* are comparable with the operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and the operator `<=>` (and have at least the comparison category type *Cat*). If no *Cat* is passed, `std::partial_ordering` is required.
- This concept is defined in the header file `<compare>`.
- Note that this concept does not imply and subsume `std::equality_comparable_with` because the latter requires that the operator `==` yields true only for two objects that are equal. With a weak or partial order, this might not be the case.
- Note that this concept does not imply and subsume std::totally_ordered_with because the latter requires that the comparison category is `std::strong_ordering` or `std::weak_ordering`.
- Requires:
  - `std::three_way_comparable` is satisfied for values and common references of *T1* and *T2* (and *Cat*)
  - All comparisons with the operators `==`, `!=`, `<`, `<=`, `>`, and `>=` yield a value convertible to `bool`
  - Any comparison with the operator `<=>` yields a comparison category (which is at least *Cat*)
  - The order of *T1* and *T2* does not matter

5.3 Concepts for Iterators and Ranges

This section lists all basic concepts for iterators and ranges, which are useful in algorithms and similar functions.

Note that the concepts for ranges are provided in the namespace `std::ranges` instead of `std`. They are declared in the header file `<ranges>`.

Concepts for iterators are declared in the header file `<iterator>`.

5.3.1 Concepts for Ranges and Views

Several concepts are defined to constrain ranges. They correspond with the concepts for iterators and deal with the fact that we have new iterator categories since C++20.

`std::ranges::`**`range`**`<`*Rg*`>`

- Guarantees that *Rg* is a valid range.
- This means that objects of type *Rg* support iteration over the elements by using `std::ranges::begin()` and `std::ranges::end()`.

  This is the case if the range is either an array or provides `begin()` and `end()` members or can be used with free-standing `begin()` and `end()` functions.

- In addition, for `std::ranges::begin()` and `std::ranges::end()`, the following constraints apply:
  - They have to operate in (amortized) constant time.
  - They do not modify the range.
  - `begin()` yields the same position when called multiple times (unless the range does not provide at least forward iterators).

  That all means that we can iterate over all elements with good performance (even multiple times unless we have pure input iterators).
- Requires:
  - For an object `rg` of type *Rg*, `std::ranges::begin(rg)` is supported and `std::ranges::end(rg)` is supported

`std::ranges::`**`output_range`**`<`*Rg*`, `*T*`>`

- Guarantees that *Rg* is a range that provides at least output iterators (iterators that you can use to write) that accept values of type *T*.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::output_iterator` is satisfied for the iterator type and `T`

`std::ranges::`**`input_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range that provides at least input iterators (iterators that you can use to read).
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::input_iterator` is satisfied for the iterator type

`std::ranges::`**`forward_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range that provides at least forward iterators (iterators that you can use to read and write and to iterate over multiple times).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::input_range<`*Rg*`>` is satisfied
  - `std::forward_iterator` is satisfied for the iterator type

`std::ranges::`**`bidirectional_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range that provides at least bidirectional iterators (iterators that you can use to read and write and to iterate over also backward).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::forward_range<`*Rg*`>` is satisfied
  - `std::bidirectional_iterator` is satisfied for the iterator type

`std::ranges::`**`random_access_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range that provides random-access iterators (iterators that you can use to read and write, jump back and forth, and compute the distance).
- Note that the `iterator_category` member of the iterators may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::bidirectional_range<`*Rg*`>` is satisfied
  - `std::random_access_iterator` is satisfied for the iterator type

`std::ranges::`**`contiguous_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range that provides random-access iterators with the additional constraint that the elements are stored in contiguous memory.
- Note that the `iterator_category` member of the iterators does not match. If the category member is defined, it is only `std::random_access_iterator_tag`, or even only `std::input_iterator_tag` if the values are prvalues.
- Requires:
  - `std::random_access_range<`*Rg*`>` is satisfied
  - `std::contiguous_iterator` is satisfied for the iterator type
  - Calling `std::ranges::data()` yields a raw pointer to the first element

`std::ranges::`**`sized_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range where the number of elements can be computed in constant time (either by calling the member `size()` or by computing the difference between the beginning and the end).
- If this concept is satisfied, `std::ranges::size()` is fast and well-defined for objects of type *Rg*.
- Note that the performance aspect of this concept is a semantic constraint, which cannot be checked at compile time. To signal that a type does not satisfy this concept even though it provides `size()`, you can define that `std::disable_sized_range<`*Rg*`>` yields `true`.[1]
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - Calling `std::ranges::size()` is supported

`std::ranges::`**`common_range`**`<`*Rg*`>`

- Guarantees that *Rg* is a range where the begin iterator and the sentinel (end iterator) have the same type.
- The guarantee is always given by:
  - All standard containers (vector, list, etc.)
  - `empty_view`
  - `single_view`
  - `common_view`

[1] The real meaning of `disable_sized_range` is “ignore size() for `sized_range`.”

The guarantee is not given by:
- take views
- `const` drop views
- iota views with no end value or an end value of a different type

For other views, it depends on the type of the underlying ranges.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - `std::ranges::iterator_t<`*Rg*`>` and `std::ranges::sentinel_t<`*Rg*`>` have the same type

`std::ranges::`**`borrowed_range`**`<`*Rg*`>`

- Guarantees that the passed range *Rg* in the current context yields iterators that can be used even when the range no longer exists. The concept is satisfied if an lvalue is passed or the passed range is always a *borrowed range*.
- If the concept is satisfied, iterators of the range are not tied to the lifetime of the range. This means that iterators cannot dangle when the range they were created from is destroyed. However, they can still dangle if the iterators of the range refer to an underlying range and the underlying range is no longer there.
- Formally, the concept is met if *Rg* is an lvalue (such as an object with a name) or if the variable template `std::ranges::enable_borrowed_range<`*Rg*`>` is `true`, which is the case for the following views: `subrange`, `ref_view`, `string_view`, `span`, `iota_view`, and `empty_view`.
- Requires:
  - `std::range<`*Rg*`>` is satisfied
  - *Rg* is an lvalue or `enable_borrowed_range<`*Rg*`>` yields `true`

`std::ranges::`**`view`**`<`*Rg*`>`

- Guarantees that *Rg* is a view (a range that is cheap to copy or move, assign, and destroy).
- A view has the following requirements:[2]
  - It has to be a range (support iteration over the elements).
  - It has to be movable.
  - The move constructor and, if available, the copy constructor have to have constant complexity.
  - The move assignment operator and, if available, the copy assignment operator have to be cheap (constant complexity or not worse than destruction plus creation).

  All but the last requirements are checked by corresponding concepts. The last requirement is a semantic constraint and has to be guaranteed by the implementer of a type by deriving publicly from `std::ranges::view_interface` or by specializing `std::ranges::enable_view<`*Rg*`>` to yield `true`.

[2] Initially, C++20 required that views should also be default constructible and that destruction should be cheap. However, these requirements were removed with `http://wg21.link/p2325r3` and `http://wg21.link/p2415r2`.

- Requires:
  - `std::range<Rg>` is satisfied
  - `std::movable<Rg>` is satisfied
  - The variable template `std::ranges::enable_view<Rg>` is `true`

`std::ranges::`**`viewable_range`**`<Rg>`

- Guarantees that *Rg* is a range that can be safely converted to a view with the `std::views::all()` adaptor.
- The concept is satisfied if *Rg* is either already a view or an lvalue of a range or a movable rvalue or a range but not an initializer list.[3]
- Requires:
  - `std::range<Rg>` is satisfied

5.3.2 Concepts for Pointer-Like Objects

This section lists all standard concepts for objects for which you can use the operator `*` to deal with a value they point to. This usually applies to raw pointers, smart pointers, and iterators. Therefore, these concepts are usually used as base constraints for concepts that deal with iterators and algorithms.

Note that support of the operator `->` is not required for these concepts.

These concepts are declared in the header file `<iterator>`.

Basic Concepts for Indirect Support

`std::`**`indirectly_writable`**`<P, Val>`

- Guarantees that *P* is a pointer-like object supporting the operator `*` to assign a *Val*.
- Satisfied by non-const raw pointers, smart pointers, and iterators provided *Val* can be assigned to where *P* refers to.

`std::`**`indirectly_readable`**`<P>`

- Guarantees that *P* is a pointer-like object supporting the operator `*` for read access.
- Satisfied by raw pointers, smart pointers, and iterators.
- Requires:
  - The resulting value has the same reference type for both `const` and non-`const` objects (which rules out `std::optional<>`). This ensures that the constness of *P* does not propagate to where it points to (which is usually not the case when the operator returns a reference to a member).
  - `std::iter_value_t<P>` must be valid. The type does ***not*** have to support the operator `->`.

[3] For lvalues of move-only view types, there is an issue that `all()` is ill-formed even though `viewable_range` is satisfied.

Concepts for Indirectly Readable Objects

For pointer-like concepts that are indirectly readable, you can check additional constraints:

`std::`**`indirectly_movable`**`<`*InP*`,`*OutP*`>`

- Guarantees that values of *InP* can be move assigned directly to values of *OutP*.
- With this concept, the following code is valid:

  ```
  void foo(InP inPos, OutP, outPos) {
    *outPos = std::move(*inPos);
  }
  ```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied
  - `std::indirectly_writable` is satisfied for rvalue references of the values of *InP* to (the values of) *OutP*

`std::`**`indirectly_movable_storable`**`<`*InP*`,`*OutP*`>`

- Guarantees that values of *InP* can be move assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.
- With this concept, the following code is valid:

  ```
  void foo(InP inPos, OutP, outPos) {
    OutP::value_type tmp = std::move(*inPos);
    *outPos = std::move(tmp);
  }
  ```

- Requires:
  - `std::indirectly_movable<`*InP*`, `*OutP*`>` is satisfied
  - `std::indirectly_writable` is satisfied for the *InP* value to the objects *OutP* refers to
  - `std::movable` is satisfied for the values *InP* refers to
  - Rvalues that *InP* refers to are copy/move constructible and assignable

`std::`**`indirectly_copyable`**`<`*InP*`, `*OutP*`>`

- Guarantees that values of *InP* can be assigned directly to values of *OutP*.
- With this concept, the following code is valid:

  ```
  void foo(InP inPos, OutP outPos) {
    *outPos = *inPos;
  }
  ```

- Requires:
  - `std::indirectly_readable<`*InP*`>` is satisfied
  - `std::indirectly_writable` is satisfied for references of the values of *InP* to (the values of) *OutP*

`std::`**`indirectly_copyable_storable`**`<`*InP*`,` *OutP*`>`

- Guarantees that values of *InP* can be assigned indirectly to values of *OutP* even when using a (temporary) object of the type to where *OutP* points to.
- With this concept, the following code is valid:

```
void foo(InP inPos, OutP outPos) {
  OutP::value_type tmp = *inPos;
  *outPos = tmp;
}
```

- Requires:
  - `std::indirectly_copyable<`*InP*`,` *OutP*`>` is satisfied
  - `std::indirectly_writable` is satisfied for `const` lvalue and rvalue references of the *InP* value to the objects *OutP* refers to
  - `std::copyable` is satisfied for the values *InP* refers to
  - The values *InP* refers to are copy/move constructible and assignable

`std::`**`indirectly_swappable`**`<`*P*`>`
`std::`**`indirectly_swappable`**`<`*P1*`,` *P2*`>`

- Guarantees that values of *P* or *P1* and *P2* can be swapped (using `std::ranges::iter_swap()`).
- Requires:
  - `std::indirectly_readable<`*P1*`>` (and `std::indirectly_readable<`*P2*`>`) is satisfied
  - For any two objects of types *P1* and *P2*, `std::ranges::iter_swap()` is supported

`std::`**`indirectly_comparable`**`<`*P1*`,` *P2*`,` *Comp*`>`
`std::`**`indirectly_comparable`**`<`*P1*`,` *P2*`,` *Comp*`,` *Proj1*`>`
`std::`**`indirectly_comparable`**`<`*P1*`,` *P2*`,` *Comp*`,` *Proj1*`,` *Proj2*`>`

- Guarantees that you can compare the elements (optionally transformed with *Proj1* and *Proj2*) to where *P1* and *P2* refer.
- Requires:
  - `std::indirect_binary_predicate` for *Comp*
  - `std::projected<`*P1*`,` *Proj1*`>` and `std::projected<`*P2*`,` *Proj2*`>` are satisfied with `std::identity` as default projections

5.3.3 Concepts for Iterators

This section lists the concepts for requiring different types of iterators. They deal with the fact that we have new iterator categories since C++20 and they correspond with the concepts for ranges.

These concepts are provided in the header file `<iterator>`.

`std::`**`input_output_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* supports the basic interface of all iterators: `operator++` and `operator*`, where `operator*` has to refer to a value.

  The concept does not require that the iterator is copyable (thus, this is less than the basic requirements for iterators used by algorithms).
- Requires:
  - `std::weakly_incrementable<`*pos*`>` is satisfied
  - The operator `*` yields a reference

`std::`**`output_iterator`**`<`*Pos*`,` *T*`>`

- Guarantees that *Pos* is an output iterator (an iterator where you can assign values to the elements) to whose values you can assign values of type*T*.
- Iterators of type *Pos* can be used to assign a value `val` of type *T* with:

```
*i++ = val;
```

- These iterators are only good for single-pass iterations.
- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_writable<`*Pos*`,` *I*`>` is satisfied

`std::`**`input_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* is an input iterator (an iterator where you can read values from the elements).
- These iterators are only good for single-pass iterations if `std::forward_iterator` is not also satisfied.
- Requires:
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::input_iterator_tag`

`std::`**`forward_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* is a forward iterator (a reading iterator with which you can iterate forward multiple times over elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::input_iterator<`*Pos*`>` is satisfied
  - `std::incrementable<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::forward_iterator_tag`

`std::`**`bidirectional_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* is a bidirectional iterator (a reading iterator with which you can iterate forward and backward multiple times over elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - Support to iterate backward with the operator `--`
  - *Pos* has an iterator category derived from `std::bidirectional_iterator_tag`

`std::`**`random_access_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* is a random-access iterator (a reading iterator with which you can jump back and forth over the elements).
- Note that the `iterator_category` member of the iterator may not match. For iterators that yield prvalues, it is `std::input_iterator_tag` (if available).
- Requires:
  - `std::bidirectional_iterator<`*Pos*`>` is satisfied
  - `std::totally_ordered<`*Pos*`>` is satisfied
  - `std::sized_sentinel_for<`*Pos*`, `*Pos*`>` is satisfied
  - Support for +, +=, −, −=, `[]`
  - *Pos* has an iterator category derived from `std::random_access_iterator_tag`

`std::`**`contiguous_iterator`**`<`*Pos*`>`

- Guarantees that *Pos* is an iterator that iterates over elements in contiguous memory.
- Note that the `iterator_category` member of the iterator does not match. If the category member is defined, it is only `std::random_access_iterator_tag`, or even only `std::input_iterator_tag` if the values are prvalues.
- Requires:
  - `std::random_access_iterator<`*Pos*`>` is satisfied
  - *Pos* has an iterator category derived from `std::contiguous_iterator_tag`
  - `to_address()` for an element is a raw pointer to the element

`std::`**`sentinel_for`**`<`*S*`, `*Pos*`>`

- Guarantees that *S* can be used as a sentinel (end iterator of maybe a different type) for *Pos*.
- Requires:
  - `std::semiregular<`*S*`>` is satisfied
  - `std::input_or_output_iterator<`*Pos*`>` is satisfied
  - Can compare *Pos* and *S* using the operators == and `!=`

`std::`**`sized_sentinel_for`**`<`*S*`, `*Pos*`>`

- Guarantees that *S* can be used as a sentinel (end iterator of maybe a different type) for *Pos* and you can compute the distance between them in constant time.
- To signal that you cannot compute the distance in constant time (even though you can compute the distance), you can define that `std::disable_sized_sentinel_for<`*Rg*`>` yields `true`.
- Requires:
  - `std::sentinel_for<`*S*`, `*Pos*`>` is satisfied
  - Calling the operator `-` for a *Pos* and *S* yields a value of the difference type of the iterator
  - `std::disable_sized_sentinel_for<`*S*`, `*Pos*`>` is not defined to yield `true`

5.3.4 Iterator Concepts for Algorithms

`std::`**`permutable`**`<`*Pos*`>`

- Guarantees that you can iterate forward by using the operator `++` and reorder elements by moving and swapping them
- Requires:
  - `std::forward_iterator<`*Pos*`>` is satisfied
  - `std::indirectly_movable_storable<`*Pos*`>` is satisfied
  - `std::indirectly_swappable<`*Pos*`>` is satisfied

`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`>`
`std::`**`mergeable`**`<`*Pos1*`, `*Pos2*`, `*ToPos*`, `*Comp*`, `*Proj1*`, `*Proj2*`>`

- Guarantees that you can merge the elements of two sorted sequences to where *Pos1* and *Pos2* refer by copying them into a sequence to where *ToPos* refers. The order is defined by the operator `<` or *Comp* (applied to the values optionally transformed with projections *Proj1* and *Proj2*).
- Requires:
  - `std::input_iterator` is satisfied for both *Pos1* and *Pos2*
  - `std::weakly_incrementable<`*ToPos*`>` is satisfied
  - `std::indirectly_copyable<`*PosN*`, `*ToPos*`>` is satisfied for both *Pos1* and *Pos2*
  - `std::indirect_strict_weak_order` of *Comp* as well as `std::projected<`*Pos1*`, `*Proj1*`>` and `std::projected<`*Pos2*`, `*Proj2*`>` are satisfied (with `<` as the default comparison and `std::identity` as the default projection), which implies:
    * `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
    * `std::copy_constructible<`*Comp*`>` is satisfied
    * `std::strict_weak_order<`*Comp*`>` is satisfied for *Comp*`&` and the (projected) value/reference types

std::**sortable**<*Pos*>
std::**sortable**<*Pos*, *Comp*>
std::**sortable**<*Pos*, *Comp*, *Proj*>

- Guarantees that you can sort the elements that the iterator *Pos* refers to with the operator < or *Comp* (after optionally applying the projection *Proj* to the values).
- Requires:
  - `std::permutable<Pos>` is satisfied
  - `std::indirect_strict_weak_order` of *Comp* and the (projected) values is satisfied (with < as the default comparison and `std::identity` as the default projection), which implies:
    - `std::indirectly_readable<Pos>` is satisfied
    - `std::copy_constructible<Comp>` is satisfied
    - `std::strict_weak_order<Comp>` is satisfied for *Comp*& and the (projected) value/reference types

5.4 Concepts for Callables

This section lists all concepts for callables

- Functions
- Function objects
- Lambdas
- Pointers to members (with the type of the object as an additional parameter)

5.4.1 Basic Concepts for Callables

std::**invocable**<*Op*, *ArgTypes...*>

- Guarantees that *Op* can be called for the arguments of types *ArgTypes...*.
- *Op* can be a function, function object, lambda, or pointer-to-member.
- To document that neither the operation nor the passed arguments are modified, you can use `std::regular_invocable` instead. However, note that there is only a semantic difference between these two concepts, which cannot be checked at compile time. Therefore, the concept differences only document the intention.
- For example:

```
struct S {
  int member;
  int mfunc(int);
  void operator()(int i) const;
};
```

```
void testCallable()
{
  std::invocable<decltype(testCallable)>      // satisfied
  std::invocable<decltype([](int){}), char>    // satisfied (char converts to int)
  std::invocable<decltype(&S::mfunc), S, int> // satisfied (member-pointer, object, args)
  std::invocable<decltype(&S::member), S>;     // satisfied (member-pointer and object)
  std::invocable<S, int>;                      // satisfied due to operator()
}
```

Note that as a type constraint, you only have to specify the parameter types:

```
void callWithIntAndString(std::invocable<int, std::string> auto op);
```

For a complete example, see the use of a lambda as a non-type template parameter.

- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`regular_invocable`**`<`*Op*`, `*ArgTypes...*`>`

- Guarantees that *Op* can be called for the arguments of types *ArgTypes...* and that the call changes neither the state of the passed operation nor of the passed arguments.
- *Op* can be a function, function object, lambda, or pointer-to-member.
- Note that the difference to the concept `std::invocable` is purely semantic and cannot be checked at compile time. Therefore, the concept differences only document the intention.
- Requires:
  - `std::invoke(op, args)` is valid for an `op` of type *Op* and `args` of type *ArgTypes...*

`std::`**`predicate`**`<`*Op*`, `*ArgTypes...*`>`

- Guarantees that the callable (function, function object, lambda) *Op* is a predicate that can be called for the arguments of types *ArgTypes...*.
- This implies that the call changes neither the state of the passed operation nor of the passed arguments.
- Requires:
  - `std::regular_invocable<`*Op*`>` is satisfied
  - All calls of *Op* with *ArgTypes...* yield a value that can be used as a as Boolean value

`std::`**`relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- Guarantees that any two objects of types *T1* and *T2* have a binary relationship in that they can be passed as arguments to the binary predicate *Pred*.
- This implies that the call changes neither the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::equivalence_relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. Therefore, the concept differences only document the intention.

- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`equivalence_relation`**`<`*Pred*`, `*T1*`, `*T2*`>`

- Guarantees that any two objects of types *T1* and *T2* have an equivalence relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is reflexive, symmetric, and transitive.
- This implies that the call changes neither the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::relation` and `std::strict_weak_order` are purely semantic and cannot be checked at compile time. Therefore, the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`strict_weak_order`**`<`*Pred*`, `*T1*`, `*T2*`>`

- Guarantees that any two objects of types *T1* and *T2* have a strict weak ordering relationship when compared with *Pred*.

  That is, they can be passed as arguments to the binary predicate *Pred* and the relationship is irreflexive and transitive.
- This implies that the call changes neither the state of the passed operation nor of the passed arguments.
- Note that the differences to the concepts `std::relation` and `std::equivalence_relation` are purely semantic and cannot be checked at compile time. Therefore, the concept differences only document the intention.
- Requires:
  - `std::predicate` is satisfied for *pred* and any combination of two objects of types *T1* and *T2*

`std::`**`uniform_random_bit_generator`**`<`*Op*`>`

- Guarantees that *Op* can be used as a random number generator that returns unsigned integral values of (ideally) equal probability.
- This concept is defined in the header file `<random>`.
- Requires:
  - `std::invocable<`*Op*`&>` is satisfied
  - `std::unsigned_integral` is satisfied for the result of the call
  - Expressions *Op*`::min()` and *Op*`::max()` are supported and yield the same type as the generator calls
  - *Op*`::min()` is less than *Op*`::max()`

5.4.2 Concepts for Callables Used by Iterators

`std::`**`indirectly_unary_invocable`**`<`*Op*`, `*Pos*`>`

- Guarantees that *Op* can be called with the values that *Pos* refers to.
- Note that the difference to the concept `indirectly_regular_unary_invocable` is purely semantic and cannot be checked at compile time. Therefore, the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The results of calling *Op* with both a value and a reference have a common reference type

`std::`**`indirectly_regular_unary_invocable`**`<`*Op*`, `*Pos*`>`

- Guarantees that *Op* can be called with the values that *Pos* refers to and the call does not change the state of *Op*.
- Note that the difference to the concept `std::indirectly_unary_invocable` is purely semantic and cannot be checked at compile time. Therefore, the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Op*`>` is satisfied
  - `std::regular_invocable` is satisfied for *Op*`&` and the value and (common) reference type of *Pos*
  - The results of calling *Op* with both a value and a reference have a common reference type

`std::`**`indirect_unary_predicate`**`<`*Pred*`, `*Pos*`>`

- Guarantees that the unary predicate *Pred* can be called with the values that *Pos* refers to.
- Requires:
  - `std::indirectly_readable<`*Pos*`>` is satisfied
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*`&` and the value and (common) reference type of *Pos*
  - All these calls of *Pred* yield a value that can be used as a Boolean value

`std::`**`indirect_binary_predicate`**`<`*Pred*`,` *Pos1*`,` *Pos2*`>`

- Guarantees that the binary predicate *Pred* can be called with the values that *Pos1* and *Pos2* refer to.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::predicate` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value that can be used as a Boolean value

`std::`**`indirect_equivalence_relation`**`<`*Pred*`,` *Pos1*`>`
`std::`**`indirect_equivalence_relation`**`<`*Pred*`,` *Pos1*`,` *Pos2*`>`

- Guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* are equivalent.
- Note that the difference to the concept `std::indirectly_strict_weak_order` is purely semantic and cannot be checked at compile time. Therefore, the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::equivalence_relation` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value that can be used as a Boolean value

`std::`**`indirect_strict_weak_order`**`<`*Pred*`,` *Pos1*`>`
`std::`**`indirect_strict_weak_order`**`<`*Pred*`,` *Pos1*`,` *Pos2*`>`

- Guarantees that the binary predicate *Pred* can be called to check whether two values of *Pos1* and *Pos1*/*Pos2* have a strict weak order.
- Note that the difference to the concept `std::indirectly_equivalence_relation` is purely semantic and cannot be checked at compile time. Therefore, the different concepts only document the intention.
- Requires:
  - `std::indirectly_readable` is satisfied for *Pos1* and *Pos2*
  - `std::copy_constructible<`*Pred*`>` is satisfied
  - `std::strict_weak_order` is satisfied for *Pred*`&`, a value or (common) reference type of *Pos1*, and a value or (common) reference type of *Pos2*
  - All these calls of *Pred* yield a value that can be used as a Boolean value

5.5 Auxiliary Concepts

This section describes some standardized concepts that are mainly specified to implement other concepts. You should usually not use them in application code.

5.5.1 Concepts for Specific Type Attributes

`std::`**`default_initializable`**`<`*T*`>`

- Guarantees that the type *T* supports default construction (declaration/construction without an initial value).
- Requires:
  - `std::constructible_from<`*T*`>` is satisfied
  - `std::destructible<`*T*`>` is satisfied
  - `T{};` is valid
  - `T x;` is valid

`std::`**`move_constructible`**`<`*T*`>`

- Guarantees that objects of type *T* can be initialized with rvalues of their type.
- That means the following operation is valid (although it might copy instead of move):

  ```
  T t2{std::move(t1)}   // for any t1 of type T
  ```

  Afterwards, `t2` shall have the former value of `t1`, which is a semantic constraint that cannot be checked at compile time.
- Requires:
  - `std::constructible_from<`*T*`,` *T*`>` is satisfied
  - `std::convertible<`*T*`,` *T*`>` is satisfied
  - `std::destructible<`*T*`>` is satisfied

`std::`**`copy_constructible`**`<`*T*`>`

- Guarantees that objects of type *T* can be initialized with lvalues of their type.
- That means the following operation is valid:

  ```
  T t2{t1}   // for any t1 of type T
  ```

  Afterwards, `t2` shall be equal to `t1`, which is a semantic constraint that cannot be checked at compile time.
- Requires:
  - `std::move_constructible<`*T*`>` is satisfied
  - `std::constructible_from` and `std::convertible_to` are satisfied for any *T*, *T*`&`, `const` *T*, and `const` *T*`&` to *T*
  - `std::destructible<`*T*`>` is satisfied

`std::`**`destructible`**`<`*T*`>`

- Guarantees that objects of type *T* are destructible without throwing an exception.

  Note that even implemented destructors automatically guarantee not to throw unless you explicitly mark them with `noexcept(false)`.
- Requires:
  - The type trait `std::is_nothrow_destructible_v<`*T*`>` yields `true`

`std::`**`swappable`**`<`*T*`>`

- Guarantees that you can swap the values of two objects of type *T*.
- Requires:
  - `std::ranges::swap()` can be called for two objects of type *T*

5.5.2 Concepts for Incrementable Types

`std::`**`weakly_incrementable`**`<`*T*`>`

- Guarantees that type *T* supports increment operators.
- Note that this concept does ***not*** require that two increments of the same value yield equal results. Therefore, this is a requirement for single-pass iterations only.
- In contrast to `std::incrementable`, the concept is also satisfied if:
  - The type is not default constructible, not copyable, or not equality comparable
  - The postfix increment operator returns `void` (or any other type)
  - Two increments of the same value give different results
- Note that the differences to concept `std::incrementable` are purely semantic differences so that types for which increments yield different results might still technically satisfy the concept `incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.
- Requires:
  - `std::default_initializable<`*T*`>` is satisfied
  - `std::movable<`*T*`>` is satisfied
  - `std::iter_difference_t<`*T*`>` is a valid signed integral type

`std::`**`incrementable`**`<`*T*`>`

- Guarantees that type *T* is an incrementable type, so that you can iterate multiple times over the same sequence of values.
- In contrast to `std::weakly_incrementable`, the concept requires that:
  - Two increments of the same value yield the same results (as is the case for forward iterators)
  - Type *T* is default constructible, copyable, and equality comparable
  - The postfix increment operator returns a copy of the iterator (has return type *T*)
- Note that the differences to concept `std::weakly_incrementable` are purely semantic differences so that types for which increments yield different results might still technically satisfy the concept

`incrementable`. To implement different behavior for this semantic difference, you should use iterator concepts instead.

- Requires:
  - `std::weakly_incrementable<`*T*`>` is satisfied
  - `std::regular<`*T*`>` is satisfied so that the type is default constructible, copyable, and equality comparable

This page is intentionally left blank

Chapter 6

Ranges and Views

Since the first C++ standard, the way to deal with the elements of containers and other sequences has always been to use iterators for the position of the first element (the *begin*) and the position behind the last element (the *end*). Therefore, algorithms that operate on ranges usually take two parameters to process all elements of a container and containers provide functions like `begin()` and `end()` to provide these parameters.

C++20 provides a new way to deal with ranges. It provides support for defining and using **ranges and subranges as single objects**, such as passing them as a whole as single arguments instead of dealing with two iterators.

The change sounds pretty simple, but as you will see, it has a lot of consequences. The way to deal with algorithms changes dramatically for both the caller and the implementor. Therefore, C++20 provides several new features and utilities for dealing with ranges:

- New overloads or variations of standard algorithms that take a range as a single argument
- Several utilities for dealing with range objects:
  - Helper functions for creating range objects
  - Helper functions for dealing with range objects
  - Helper types for dealing with range objects
  - Concepts for ranges
- Lightweight ranges, called ***views***, to refer to (a subset of) a range with optional transformation of the values
- ***Pipelines*** as a flexible way to compose the processing of ranges and views

This chapter introduces the basic aspects and features of ranges and views. The following chapters discuss details.

6.1 A Tour of Ranges and Views Using Examples

Let us look at the role of ranges and views by looking at a few examples that use them and discussing the basic principles without going into too much detail yet.

6.1.1 Passing Containers to Algorithms as Ranges

Since the first C++ standard published in C++98, we iterate over half-open ranges when dealing with collections of elements. By passing the begin and the end of a range (often coming from the `begin()` and `end()` member functions of a container), you can specify which elements have to be processed:

```
#include <vector>
#include <algorithm>

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::sort(coll.begin(), coll.end());    // sort all elements of the collection
```

C++20 introduces the concept of a ***range***, which is a single object that represents a sequence of values. Any container can be used as such a range.

As a consequence, you can now pass containers as a whole to algorithms:

```
#include <vector>
#include <algorithm>

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::ranges::sort(coll);                // sort all elements of the collection
```

Here, we pass the vector `coll` to the range algorithm `sort()` to sort all elements in the vector.

Most standard algorithms support passing ranges as one argument since C++20. However, there is no support for parallel algorithms and numeric algorithms (yet). Besides taking only one argument for a range as a whole, the new algorithms may have a couple of minor differences:

- They use concepts for iterators and ranges to ensure that you pass valid iterators and ranges.
- They might have different return types.
- They might return borrowed iterators, signaling that there is no valid iterator because a temporary range (rvalue) was passed.

For details, see the overview of algorithms in C++20.

Namespace for Ranges

The new algorithms of the ranges library such as `sort()` are provided in the namespace `std::ranges`. In general, the C++ standard library provides all features for dealing with ranges in special namespaces:

- Most of them are provided in the namespace **`std::ranges`**.
- A few of them are provided in **`std::views`**, which is an alias for **`std::ranges::views`**.

New namespaces were necessary because the ranges library introduces several new APIs that use the same symbol names. However, C++20 should not break existing code. Therefore, the namespaces were necessary to avoid ambiguities and other conflicts with existing APIs.

It is sometimes surprising to see what belongs to the ranges library and its namespace `std::ranges` and what belongs to `std`. Roughly speaking, `std::ranges` is used for utilities that deal with ranges as a whole. For example:

- Some concepts belong to `std`, some to `std::ranges`.
  For example, for iterators, we have the concept `std::forward_iterator`, but the corresponding concept for ranges is `std::ranges::forward_range`.
- Some type functions belong to `std`, some to `std::ranges`.
  For example, we have the type trait `std::iter_value_t`, but the corresponding type trait for ranges is `std::ranges::range_value_t`.
- Some symbols are even provided in both `std` and `std::ranges`.
  For example, we have the free-standing functions `std::begin()` and `std::ranges::begin()`.

If both can be used, it is **better to use symbols and utilities of the namespace `std::ranges`**. The reason for this is that the new utilities of the ranges library may fix flaws that the older utilities have. For example, it is better to use `begin()` or `cbegin()` from the namespace `std::ranges`.

It is pretty common to introduce a shortcut for the namespace `std::ranges`, such as `rg` or `rng`. Therefore, code above can also look as follows:

```
#include <vector>
#include <algorithm>
namespace rg = std::ranges;              // define shortcut for std::ranges

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

rg::sort(coll);                          // sort all elements of the collection
```

Do ***not*** use a `using namespace` directive to avoid having to qualify range symbols at all. Otherwise, you can easily end up with code that has compile-time conflicts or uses wrong lookups meaning that nasty bugs can occur.

Header Files for Ranges

Many of the new features of the ranges library are provided in a new header file **`<ranges>`**. However, some of them are provided in existing header files. One example is the range algorithms, which are declared in `<algorithm>`.

Therefore, to use the algorithms provided for ranges as single arguments, you still only have to include `<algorithm>`.

However, for a couple of additional features provided by the ranges library, you need the header file `<ranges>`. For this reason, you should always include `<ranges>` when using something from the namespaces `std::ranges` or `std::views`:

```
#include <vector>
#include <algorithm>
```

```
#include <ranges>                    // for ranges utilities (so far not necessary yet)

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::ranges::sort(coll);             // sort all elements of the collection
```

6.1.2 Constraints and Utilities for Ranges

The new standard algorithms for ranges declare range parameters as template parameters (there is no common type you can use for them). To specify and validate the necessary requirements when dealing with these range parameters, C++20 introduces several range concepts. In addition, utilities help you to apply these concepts.

Consider, for example, the `sort()` algorithm for ranges. In principle, it is defined as follows (a few details are missing and will be introduced later):

```
template<std::ranges::random_access_range R,
         typename Comp = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<R>, Comp>
... sort(R&& r, Comp comp = {});
```

This declaration already has multiple new features of ranges:

- Two standard concepts specify requirements for the passed range `R`:
  - The concept **`std::ranges::random_access_range`** requires that `R` is a range that provides random-access iterators (iterators that you can use to jump back and forth between elements and compute their distance). The concept includes (subsumes) the basic concept for ranges: `std::range`, which requires that for the passed argument, you can iterate over elements from `begin()` to `end()` (at least using `std::ranges::begin()` and `std::ranges::end()`, which means that this concept is also satisfied by raw arrays).
  - The concept **`std::sortable`** requires that the elements in the range `R` can be sorted with the sort criterion `Comp`.
- The new type utility `std::ranges::iterator_t` is used to pass the iterator type to `std::sortable`.
- `Comp std::ranges::less` is used as a default comparison criterion. It defines that the sort algorithm sorts the elements with the operator `<`. `std::ranges::less` is kind of a concept-constrained `std::less`. It ensures that all comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`) are supported and that the values have a total ordering.

This means that you can pass any range with random-access iterators and sortable elements. For example:

ranges/rangessort.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> coll{"Rio", "Tokyo", "New York", "Berlin"};

  std::ranges::sort(coll);       // sort elements
  std::ranges::sort(coll[0]);    // sort character in first element
  print(coll);

  int arr[] = {42, 0, 8, 15, 7};
  std::ranges::sort(arr);        // sort values in array
  print(arr);
}
```

The program has the following output:

```
Beilnr New York Rio Tokyo
0 7 8 15 42
```

If you pass a container/range that has no random-access iterators, you get an error message stating that the concept `std::ranges::random_access_range` is not satisfied:

```
std::list<std::string> coll2{"New York", "Rio", "Tokyo"};
std::ranges::sort(coll2);   // ERROR: concept random_access_range not satisfied
```

If you pass a container/range where you cannot compare the elements with the operator <, `std::sortable` is not satisfied:

```
std::vector<std::complex<double>> coll3;
std::ranges::sort(coll3);   // ERROR: concept sortable not satisfied
```

Range Concepts

Table *Basic range concepts* lists the basic concepts to define requirements for ranges.

Concept (in `std::ranges`)	**Requires**
`range`	Can be iterated from begin to end
`output_range`	Range of elements to write values to
`input_range`	Range to read element values from
`forward_range`	Range you can iterate over the elements multiple times
`bidirectional_range`	Range you can iterate over the elements forward and backward
`random_access_range`	Range with random access (jump back and forth between elements)
`contiguous_range`	Range with all elements in contiguous memory
`sized_range`	Range with constant-time `size()`

Table 6.1. Basic range concepts

Note the following:

- `std::ranges::range` is the base concept of all other range concepts (all other concepts subsume this concept).
- `output_range` as well as the hierarchy of `input_range`, `forward_range`, `bidirectional_range`, `random_access_range`, and `contiguous_range` map to corresponding iterator categories and build a corresponding subsumption hierarchy.
- `std::ranges::contiguous_range` is a new category of ranges/iterators for which it is guaranteed that the elements are stored in contiguous memory. Code can benefit from this fact by using raw pointers to iterate over the elements.
- `std::ranges::sized_range` is independent from other constraints except that it is a `range`.

Note that the iterator and corresponding range categories changed slightly with C++20. In fact, the C++ standard now supports two versions of iterator categories: those before C++20 and those since C++20, which might not be the same.

Table *Other range concepts* lists a few other range concepts for special cases that are introduced later.

Concept (in `std::ranges`)	**Requires**
`view`	Range that is cheap to copy or move and assign
`viewable_range`	Range that can be converted to a view (with `std::ranges::all()`)
`borrowed_range`	Range with iterators that are not tied to the lifetime of the range
`common_range`	Ranges where begin and end (sentinel) have the same type

Table 6.2. Other range concepts

For details, see the discussion of range concepts.

6.1.3 Views

To deal with ranges, C++20 also introduces ***views***. Views are lightweight ranges that are cheap to create and copy/move: Views can:

- Refer to ranges and subranges
- Own temporary ranges
- Filter out elements
- Yield transformed values of the elements
- Generate a sequence of values themselves

Views are usually used to process, on an ad-hoc basis, a subset of the elements of an underlying range and/or their values after some optional transformation. For example, you can use a view to iterate over only the first five elements of a range as follows:

```
for (const auto& elem : std::views::take(coll, 5)) {
  ...
}
```

`std::views::take()` is a ***range adaptor*** for creating a view that operates on the passed range `coll`. In this case, `take()` creates a view to the first *n* elements of a passed range (if there are any). So with

```
std::views::take(coll, 5)
```

we pass a view to `coll` that ends with its sixth element (or the last element if there are fewer elements).

The C++ standard library provides several range adaptors and factories for creating views in the namespace `std::views`. The views created provide the usual API of ranges so that `begin()`, `end()`, and the operator `++` can be used to iterate over the elements and the operator `*` can be used to deal with the values.

Pipelines of Ranges and Views

There is an alternative syntax for calling range adaptors that operate on a range passed as a single argument:

```
for (const auto& elem : coll | std::views::take(5)) {
  ...
}
```

Both forms are equivalent. However, the pipeline syntax makes it more convenient to create a sequence of views on a range, which we will discuss in detail in a moment. That way, simple views can be used as building blocks for more complex processing of collections of elements.

Assume you want to use the following three views:

```
// view with elements of coll that are multiples of 3:
std::views::filter(coll, [] (auto elem) {
                             return elem % 3 == 0;
                           })

// view with squared elements of coll:
std::views::transform(coll, [] (auto elem) {
                                return elem * elem;
                              })

// view with first three elements of coll:
std::views::take(coll, 3)
```

Because a view is a range, you can use a view as an argument of another view:

```
// view with first three squared values of the elements in coll that are multiples of 3:
auto v = std::views::take(
           std::views::transform(
             std::views::filter(coll,
                                [] (auto elem) { return elem % 3 == 0; }),
             [] (auto elem) { return elem * elem; }),
           3);
```

This nesting is hard to read and maintain. Here, however, we can benefit from the alternative pipe syntax to let a view operate on a range. By using the operator `|`, we can create ***pipelines*** of views:

```
// view with first three squared values of the elements in coll that are multiples of 3:
auto v = coll
           | std::views::filter([] (auto elem) { return elem % 3 == 0; })
           | std::views::transform([] (auto elem) { return elem * elem; })
           | std::views::take(3);
```

This *pipeline* of ranges and views is easy to define and to understand.

For a collection such as

```
std::vector coll{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

the output would be:

```
9 36 81
```

Here is another full program that demonstrates the composability of views with the pipeline syntax:

ranges/viewspipe.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <ranges>

int main()
{
  namespace vws = std::views;

  // map of composers (mapping their name to their year of birth):
  std::map<std::string, int> composers{
    {"Bach", 1685},
    {"Mozart", 1756},
    {"Beethoven", 1770},
    {"Tchaikovsky", 1840},
    {"Chopin", 1810},
    {"Vivaldi ", 1678},
  };

  // iterate over the names of the first three composers born since 1700:
  for (const auto& elem : composers
                          | vws::filter([](const auto& y) {    // since 1700
                                return y.second >= 1700;
                              })
                          | vws::take(3)                       // first three
                          | vws::keys                          // keys/names only
                          ) {
    std::cout << "- " << elem << '\n';
  }
}
```

In this example, we apply a couple of views to a map of composers, where the elements have their name and their year of birth (note that we introduce `vws` as a shortcut for `std::views`):

```
std::map<std::string, int> composers{ ... };
...
composers
 | vws::filter(...)
 | vws::take(3)
 | vws::keys
```

The composing pipeline is passed directly to the range-based `for` loop and produces the following output (remember that the elements in the map are sorted according to their keys/names):

```
- Beethoven
- Chopin
- Mozart
```

Generating Views

Views can also generate a sequence of values themselves. For example, by using the iota view, we can iterate over all values from 1 to 10 as follows:

```
for (int val : std::views::iota(1, 11)) {         // iterate from 1 to 10
  ...
}
```

Type and Lifetime of Views

You can create the views and give them a name before you use them:

```
auto v1 = std::views::take(coll, 5);
auto v2 = coll | std::views::take(5);
...
for (int val : v1) {
  ...
}
std::ranges::sort(v2);
```

However, most views on lvalues (ranges that have a name) have reference semantics. Therefore, you have to ensure that the referred ranges and iterators of views still exist and are valid.

You should use `auto` to declare the views because the exact type can be tricky to specify. In this case, for example, for `coll` having type `std::vector<int>`, both `v1` and `v2` have the following type:

```
std::ranges::take_view<std::ranges::ref_view<std::vector<int>>>
```

Internally, adaptors and constructors may create nested view types such as a `std::ranges::take_view` or `std::ranges::iota_view` referring to a `std::ranges::ref_view` that is used to refer to the elements of the passed external container.

You could also declare and initialize the real view directly. However, usually you should use the adaptors and factories provided to create and initialize views. Using the adaptors and factories is usually better because they are easier to use, are often smarter, and may provide optimizations. For example, `take()` might yield just a `std::string_view` if the range passed is already a `std::string_view`.

You might now be wondering whether all these view types create significant overhead in code size and running time. Note that the view types use only small or trivial inline functions so that, for usual cases at least, optimizing compilers can avoid significant overhead.

Views for Writing

Views on lvalues usually have reference semantics. This means that, in principle, views can be used for both reading and writing.

For example, we can sort only the first five elements of `coll` as follows:

```
std::ranges::sort(std::views::take(coll, 5));     // sort the first five elements of coll
```

or as follows:

```
std::ranges::sort(coll | std::views::take(5));    // sort the first five elements of coll
```

This means that usually:

- If elements of the referred range are modified, the elements of the view were modified.
- If elements of the view are modified, the elements of the referred range were modified.

Lazy Evaluation

Besides having reference semantics, views use lazy evaluation. This means that views do their processing for the next element when an iterator to the view calls `begin()` or `++` or the value of the element is requested:

```
auto v = coll | std::views::take(5);   // neither goes to the first element nor to its value
...
auto pos = v.begin();                  // goes to the first element
...
std::cout << *pos;                     // goes to its value
...
++pos;                                 // goes to the next element
...
std::cout << *pos;                     // goes to its value
```

Caching

In addition, some views use caching. If going to the first element of a view with `begin()` needs some computation (because we skip leading elements), a view might cache the return value of `begin()` so that the next time we call `begin()`, we do not have to compute the position of the first element again.

However, this optimization has significant consequences:

- You might not be able to iterate over views when they are `const`.
- Concurrent iterations can cause undefined behavior even if we do not modify anything.
- Early read access might invalidate or change the later iterations over elements of views.

We will discuss all of this later in detail. For the moment, note that the **standard views are considered harmful** when modifications happen:

- Inserting or removing elements in the underlying range may have significant impact on the functionality of a view. After such modification, a view might behave differently or even no longer be valid.

Therefore, it is strongly recommended to use views right before you need them. **Create views to use them ad hoc.** If modifications happen between initializing a view and using it, care has to be taken.

6.1.4 Sentinels

To deal with ranges, we have to introduce a new term, ***sentinels***, which represent the end of a range.

In programming, a *sentinel* is a special value that marks an end or termination. Typical examples are:

- The null terminator `'\0'` as the end of a character sequence (e.g., used in string literals)
- `nullptr` marking the end of a linked list
- `-1` to mark the end of a list of non-negative integers

In the ranges library, sentinels define the **end of a range**. In the traditional approach of the STL, sentinels would be the *end iterators*, which usually have the same type as the iterators that iterate over a collection. However, with C++20 ranges, having the same type is no longer required.

The requirement that end iterators have to have the same type as the iterators that define the begin of a range and that are used to iterate over the elements causes some drawbacks. Creating an end iterator may be expensive or might not even be possible:

- If we want to use a C string or string literal as a range, we first have to compute the end iterator by iterating over the characters until we find `'\0'`. Thus, before we use the string as a range, we have already done the first iteration. Dealing then with all the characters would need a second iteration.
- In general, this principle applies if we define the end of a range with a certain value. If we need an end iterator to deal with the range, we first have to iterate over the whole range to find its end.
- Sometimes, iterating twice (once to find the end and then once to process the elements of the range) is not possible. This applies to ranges that use pure input iterators, such as using input streams that we read from as ranges. To compute the end of the input (may be *EOF*), we already have to read the input. Reading the input again is not possible or will yield different values.

The generalization of end iterators as sentinels solves this dilemma. C++20 ranges support sentinels (end iterators) of different types. They can signal "until `'\0'`", "until EOF", or until any other value. They can even signal "there is no end" to define endless ranges and "hey, iterating iterator, check yourself whether there is the end."

Note that before C++20, we could also have these kinds of sentinels but they were required to have the same type as the iterators. One example was *input stream iterators*: a default constructed iterator of type `std::istream_iterator<>` was used to create an *end-of-stream iterator* so that you could process input from a stream with an algorithm until end-of-file or an error occurs:

```
// print all int values read from standard input:
std::for_each(std::istream_iterator<int>{std::cin},  // read ints from cin
              std::istream_iterator<int>{},          // end is an end-of-file iterator
              [] (int val) {
                std::cout << val << '\n';
              });
```

By relaxing the requirement that sentinels (end iterators) now have to have the same type as iterating iterators, we gain a couple of benefits:

- We can skip the need to find the end before we start to process. We do both together: process the values and find the end while iterating.
- For the end iterator, we can use types that disable operations that cause undefined behavior (such as calling the operator `*`, because there is no value at the end). We can use this feature to signal an error at compile time when we try to dereference an end iterator.
- Defining an end iterator becomes easier.

Let us look at a simple example using a sentinel of a different type to iterate over a "range" where the types of the iterators differ:

ranges/sentinel1.cpp

```
#include <iostream>
#include <compare>
#include <algorithm>  // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';  // end is where iterator points to '\0'
  }
};

int main()
{
  const char* rawString = "hello world";

  // iterate over the range of the begin of rawString and its end:
  for (auto pos = rawString; pos != NullTerm{}; ++pos) {
    std::cout << ' ' << *pos;
  }
  std::cout << '\n';

  // call range algorithm with iterator and sentinel:
  std::ranges::for_each(rawString,   // begin of range
                                 NullTerm{},   // end is null terminator
                                 [] (char c) {
                                   std::cout << ' ' << c;
                                 });
  std::cout << '\n';
}
```

The program has the following output:

```
h e l l o   w o r l d
h e l l o   w o r l d
```

In the program, we first define an end iterator that defines the end as having a value equal to `'\0'`:

```
struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';   // end is where iterator points to '\0'
  }
};
```

Note that we combine a few other new features of C++20 here:

- When defining the member function `operator==()`, we use `auto` as the parameter type. That way we make the member function generic so that the operator `==` can compare with an object `pos` of arbitrary type (provided comparing the values that `pos` refers to with `'\0'` is valid).
- We define only the operator `==` as a generic member function even though algorithms usually compare iterators with sentinels using the operator `!=`. Here, we benefit from the fact that C++20 can now map the operator `!=` to the operator `==` with an arbitrary order of the operands.

Using Sentinels Directly

We then use a basic loop to iterate over the "range" of characters of the string `rawString`:

```
for (auto pos = rawString; pos != NullTerm{}; ++pos) {
  std::cout << ' ' << *pos;
}
```

We initialize `pos` as an iterator that iterates over the characters and prints out their values with `*pos`. The loop runs while its comparison with `NullTerm{}` yields that the value of `pos` does not equal `'\0'`. That way, `NullTerm{}` serves as a sentinel. It does not have the same type as `pos` but it supports a comparison with `pos` in such a way that it checks the current value that `pos` refers to.

Here you can see how sentinels are the generalization of end iterators. They may have a different type than the iterator that iterates over the elements, but they support comparisons with the iterator to decide whether we are at the end of a range.

Passing Sentinels to Algorithms

C++20 provides overloads for algorithms that no longer require that the begin iterator and the sentinel (end iterator) have the same type. However, these overloads are provided in the namespace `std::ranges`:

```
std::ranges::for_each(rawString,     // begin of range
                      NullTerm{},    // end is null terminator
                      ... );
```

The algorithms in the namespace `std` still require that the begin iterator and the end iterator have the same type and cannot be used in that way:

```
std::for_each(rawString, NullTerm{},   // ERROR: begin and end have different types
              ... );
```

If you have two iterators of different types, type `std::common_iterator` provides a way to harmonize them for traditional algorithms. This can be useful because numeric algorithms, parallel algorithms, and containers still require that the begin and the end iterator have the same type.

6.1.5 Range Definitions with Sentinels and Counts

Ranges can be more than just containers or a pair of iterators. Ranges can be defined by:

- A begin iterator and an end iterator of the same type
- A begin iterator and a sentinel (an end marker of maybe a different type)
- A begin iterator and a count
- Arrays

The ranges library supports the definition and use of all of these ranges.

First, the algorithms are implemented in such a way that the ranges can be arrays. For example:

```
int rawArray[] = {8, 6, 42, 1, 77};
...
std::ranges::sort(rawArray);   // sort elements in the raw array
```

In addition, there are several utilities for defining ranges defined by iterators and sentinels or counts, and these are introduced in the following subsections.

Subranges

To define ranges of iterators and sentinels, the ranges library provides type `std::ranges::`**`subrange<>`**.

Let us look at a simple example using subranges:

ranges/sentinel2.cpp

```
#include <iostream>
#include <compare>
#include <algorithm>   // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';   // end is where iterator points to '\0'
  }
};
```

```
int main()
{
  const char* rawString = "hello world";

  // define a range of a raw string and a null terminator:
  std::ranges::subrange rawStringRange{rawString, NullTerm{}};

  // use the range in an algorithm:
  std::ranges::for_each(rawStringRange,
                        [] (char c) {
                          std::cout << ' ' << c;
                        });
  std::cout << '\n';

  // range-based for loop also supports iterator/sentinel:
  for (char c : rawStringRange) {
    std::cout << ' ' << c;
  }
  std::cout << '\n';
}
```

As introduced as an example of a sentinel we define type `NullTerm` as a type for sentinels that check for the null terminator of a string as the end of a range.

By using a `std::ranges::subrange`, the program defines a range object that represents the beginning of the string and the sentinel as its end:

```
std::ranges::subrange rawStringRange{rawString, NullTerm{}};
```

A subrange is *the* generic type that can be used to convert a range defined by an iterator and a sentinel into a single object that represents this range. In fact, the range is even a view, which internally, just stores the iterator and the sentinel. This means that subranges have reference semantics and are cheap to copy.

As a subrange, we can pass the range to the new algorithms that take ranges as single arguments:

```
std::ranges::for_each(rawStringRange, ... );      // OK
```

Note that subranges are not always common ranges, meaning that calling `begin()` and `end()` for them may yield different types. Subranges just yield what was was passed to define the range.

Even when a subrange is not a common range, you can pass it to a range-based `for` loop. The range-based `for` loop accepts ranges where the types of the begin iterator and the sentinel (end iterators) differ (this feature was already introduced with C++17, but with ranges and views, you can really benefit from it):

```
for (char c : std::ranges::subrange{rawString, NullTerm{}}) {
  std::cout << ' ' << c;
}
```

We can make this approach even more generic by defining a class template where you can specify the value that ends a range. Consider the following example:

ranges/sentinel3.cpp

```
#include <iostream>
#include <algorithm>

template<auto End>
struct EndValue {
  bool operator== (auto pos) const {
    return *pos == End;   // end is where iterator points to End
  }
};

int main()
{
  std::vector coll = {42, 8, 0, 15, 7, -1};

  // define a range referring to coll with the value 7 as end:
  std::ranges::subrange range{coll.begin(), EndValue<7>{}};

  // sort the elements of this range:
  std::ranges::sort(range);

  // print the elements of the range:
  std::ranges::for_each(range,
                        [] (auto val) {
                          std::cout << ' ' << val;
                        });
  std::cout << '\n';

  // print all elements of coll up to -1:
  std::ranges::for_each(coll.begin(), EndValue<-1>{},
                        [] (auto val) {
                          std::cout << ' ' << val;
                        });
  std::cout << '\n';
}
```

Here, we define `EndValue<>` as the end iterator, checking for the end passed as a template parameter. `EndValue<7>{}` creates an end iterator where 7 ends the range and `EndValue<-1>{}` creates an end iterator where `-1` ends the range.

The output of the program is as follows:

```
 0 8 15 42
 0 8 15 42 7
```

You could define a value of any supported non-type template parameter type.

As another example of sentinels, look at `std::unreachable_sentinel`. This is a value that C++20 defines to represent the "end" of an endless range. It can help you to optimize code so that it never compares against the end (because that comparison is useless if it always yields `false`).

For more aspects of subranges, see the details of subranges.

Ranges of Begin and a Count

The ranges library provides multiple ways of dealing with ranges defined as beginning and a count.

The most convenient way to create a range with a begin iterator and a count is to use the range adaptor `std::views::counted()`. It creates a cheap view to the first *n* elements of a begin iterator/pointer.

For example:

```
std::vector<int> coll{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto pos5 = std::ranges::find(coll, 5);
if (std::ranges::distance(pos5, coll.end()) >= 3) {
  for (int val : std::views::counted(pos5, 3)) {
    std::cout << val << ' ';
  }
}
```

Here, `std::views::counted(pos5, 3)` creates a view that represents the three elements starting with the element to which `pos5` refers. Note that `counted()` does ***not*** check whether there are elements (passing a count that is too high results in undefined behavior). It is up to the programmer to ensure that the code is valid. Therefore, with `std::ranges::distance()`, we check whether there are enough elements (note that this check can be expensive if your collection does not have random-access iterators).

If you know that there is an element with the value `5` that has at least two elements behind it, you can also write:

```
// if we know there is a 5 and at least two elements behind:
for (int val : std::views::counted(std::ranges::find(coll, 5), 3)) {
  std::cout << val << ' ';
}
```

The count may be `0`, which means that the range is empty.

Note that you should use `counted()` only when you really have an iterator and a count. If you already have a range and want to deal with the first *n* elements only, use `std::views::take()`.

For more details, see the description of `std::views::counted()`.

6.1.6 Projections

`sort()` and many other algorithms for ranges usually have an additional optional template parameter, a *projection*:

```
template<std::ranges::random_access_range R,
         typename Comp = std::ranges::less,
         typename Proj = std::identity>
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
... sort(R&& r, Comp comp = {}, Proj proj = {});
```

The optional additional parameter allows you to specify a transformation (*projection*) for each element before the algorithm processes it further.

For example, `sort()` allows you to specify a projection for the elements to be sorted separately from the way the resulting values are compared:

```
std::ranges::sort(coll,
                  std::ranges::less{},    // still compare with <
                  [] (auto val) {         // but use the absolute value
                    return std::abs(val);
                  });
```

This might be more readable or easier to program than:

```
std::ranges::sort(coll,
                  [] (auto val1, auto val2) {
                    return std::abs(val1) < std::abs(val2);
                  });
```

See *ranges/rangesproj.cpp* for a complete example.

The default projection is `std::identity()`, which simply yields the argument passed to it so that it performs no projection/transformation at all. (`std::identity()` is defined as a new function object in `<functional>`).

User-defined projections simply have to take one parameter and return a value for the transformed parameter.

As you can see, the requirement that the elements have to be sortable takes the projection into account:

```
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
```

6.1.7 Utilities for Implementing Code for Ranges

To make it easy to program against all the different kinds of ranges, the ranges library provides the following utilities:

- **Generic functions** that, for example, yield iterators or the size of a range
- **Type functions** that, for example, yield the type of an iterator or the type of the elements

Assume that we want to implement an algorithm that yields the maximum value in a range:

ranges/maxvalue1.hpp

```
#include <ranges>

template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
}
```

Here, we use a couple of standard utilities to deal with the range `rg`:

- The concept `std::ranges::input_range` to require that the passed parameter is a range that we can read from
- The type function `std::ranges::range_value_t` that yields the type of the elements in the range
- The helper function `std::ranges::empty()` that yields whether the range is empty
- The helper function `std::ranges::begin()` that yields an iterator to the first element (if there is one)
- The helper function `std::ranges::end()` that yields a *sentinel* (end iterator) of the range

With the help of these type utilities, the algorithm even works for all ranges (including arrays) because the utilities are also defined for them.

For example, `std::ranges::empty()` tries to call a member function `empty()`, a member function `size()`, a free-standing function `size()`, or to check whether the begin iterator and the sentinel (end iterators) are equal. The utility functions for ranges are documented later in detail.

Note that the generic `maxValue()` function should declare the passed range `rg` as a ***universal reference*** (also called a *forwarding reference*) because you cannot iterate over some lightweight ranges (views) when they are `const`:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

This is discussed later in detail.

6.1.8 Limitations and Drawbacks of Ranges

In C++20, ranges also have some major limitations and drawbacks that should be mentioned in a general introduction to them:

- There is no ranges support for numeric algorithms yet. To pass a range to a numeric algorithm, you have to pass `begin()` and `end()` explicitly:

  ```
  std::ranges::accumulate(cont, 0L);                     // ERROR: not provided
  std::accumulate(cont.begin(), cont.end(), 0L);         // OK
  ```

- There is no support for ranges for parallel algorithms yet.

  ```
  std::ranges::sort(std::execution::par, cont);              // ERROR: not provided
  std::sort(std::execution::par, cont.begin(), cont.end());  // OK
  ```

  However, note that you should be careful when using the existing parallel algorithms for views by passing `begin()` and `end()` to them. For some views, concurrent iterations cause undefined behavior. Only do this after declaring the view to be `const`.
- Several traditional APIs for ranges defined by a begin iterator and an end iterator require that the iterators have the same type (e.g., this applies to containers and the algorithms in the namespace `std`). You might have to harmonize their types with `std::views::common()` or `std::common_iterator`.
- For some views, you cannot iterate over elements when the views are `const`. Generic code might therefore have to use universal/forwarding references.
- `cbegin()` and `cend()` functions, which were designed to ensure that you cannot (accidentally) modify the element you iterate over, are broken for views that refer to non-`const` objects.
- When views refer to containers, their propagation of constness may be broken.
- Ranges lead to a significant namespace mess. For example, look at the following declaration of `std::find()` but with all standard names fully qualified:

  ```
  template<std::ranges::input_range Rg,
           typename T,
           typename Proj = std::identity>
  requires std::indirect_binary_predicate<std::ranges::equal_to,
                                          std::projected<std::ranges::iterator_t<Rg>, Proj>,
                                          const T*>
  constexpr std::ranges::borrowed_iterator_t<Rg>
    find(Rg&& rg, const T& value, Proj proj = {});
  ```

  It is really not easy to know which namespace has to be used where.

  In addition, we have some symbols in both the namespace `std` and the namespace `std::ranges` with slightly different behavior. In the declaration above, `equal_to` is such an example. You could also use `std::equal_to`, but in general, utilities in `std::ranges` provide better support and are more robust for corner cases.

6.2 Borrowed Iterators and Ranges

When passing a range as a single argument to an algorithm, you get lifetime problems. This section describes how the ranges library deals with this issue.

6.2.1 Borrowed Iterators

Many algorithms return iterators to the ranges they operate on. However, when passing ranges as a single argument, we can get a new problem that was not possible when a range required two arguments (the begin iterator and the end iterator): if you pass a temporary range (such as a range returned by a function) and return an iterator to it, the returned iterator might become invalid at the end of the statement when the range is destroyed. Using the returned iterator (or a copy of it) would result in undefined behavior.

For example, consider passing a temporary range to a `find()` algorithm that searches for a value in a range:

```
std::vector<int> getData();     // forward declaration

auto pos = find(getData(), 42);                // returns iterator to temporary vector
// temporary vector returned by getData() is destroyed here
std::cout << *pos;                             // OOPS: using a dangling iterator
```

The return value of `getData()` is destroyed with the end of the statement where it is used. Therefore, `pos` refers to an element of a collection that is no longer there. Using `pos` causes undefined behavior (at best, you get a core dump so that you can see that you have a problem).

To deal with this problem, the ranges library has introduced the concept of *borrowed iterators*. A borrowed iterator ensures that its lifetime does not depend on a temporary object that might have been destroyed. If it does, using it results in a compile-time error. Thus, a borrowed iterator signals whether it can safely outlive the passed range, which is the case if the range was not temporary or the state of the iterator does not depend on the state of the passed range. If you have a borrowed iterator that refers to a range, the iterator is safe to use and does not dangle even when the range is destroyed.[1]

By using type `std::ranges::borrowed_iterator_t<>`, algorithms can declare the returned iterator as *borrowed*. This means that the algorithm always returns an iterator that is safe to use after the statement. If it could dangle, a special return value is used to signal this and convert possible runtime errors into compile-time errors.

For example, `std::ranges::find()` for a single range is declared as follows:

```
template<std::ranges::input_range Rg,
         typename T,
         typename Proj = identity>
...
constexpr std::ranges::borrowed_iterator_t<Rg>
  find(Rg&& r, const T& value, Proj proj = {});
```

By specifying the return type as `std::ranges::borrowed_iterator_t<>` of `Rg`, the standard enables a compile-time check: if the range *R* passed to the algorithm is a temporary object (a prvalue), the return type becomes a *dangling iterator*. In that case, the return value is an object of type `std::ranges::dangling`. Any use of such an object (except copying and assignment) results in a compile-time error.

[1] For this reason, such iterators were called *safe iterators* in draft versions of the ranges library.

Therefore, the following code results in a compile-time error:

```
std::vector<int> getData();     // forward declaration

auto pos = std::ranges::find(getData(), 42);   // returns iterator to temporary vector
// temporary vector returned by getData() was destroyed
std::cout << *pos;                             // compile-time ERROR
```

To be able to call `find()` for a temporary, you have to pass it as an lvalue. That is, it has to have a name. That way, the algorithm ensures that the collection still exists after calling it. It also means that you can also check whether a value was found (which is usually appropriate anyway).

The best approach to give a returned collection a name is to bind it to a reference. That way, the collection is never copied. Note that by rule, references to temporary objects always extend their lifetime:

```
std::vector<int> getData();           // forward declaration

reference data = getData();           // give return value a name to use it as an lvalue
// lifetime of returned temporary vector ends now with destruction of data
...
```

The are two kinds of references you can use here:

- You can declare a `const` lvalue reference:

  ```
  std::vector<int> getData();                  // forward declaration

  const auto& data = getData();                // give return value a name to use it as an lvalue
  auto pos = std::ranges::find(data, 42);      // yields no dangling iterator
  if (pos != data.end()) {
    std::cout << *pos;                         // OK
  }
  ```

  This reference makes the return value `const`, which might not be what you want (note that you cannot iterate over some views when they are `const`; although, due to the reference semantics of views, you have to be careful when returning them).

- In more generic code, you should use a *universal reference* (also called a *forwarding reference*) or `decltype(auto)` so that you keep the natural non-constness of the return value:

  ```
  ... getData();                               // forward declaration

  auto&& data = getData();                     // give return value a name to use it as an lvalue
  auto pos = std::ranges::find(data, 42);      // yields no dangling iterator
  if (pos != data.end()) {
    std::cout << *pos;                         // OK
  }
  ```

The consequence of this feature is that you cannot pass a temporary object to an algorithm even if the resulting code would be valid:

```
process(std::ranges::find(getData(), 42));    // compile-time ERROR
```

Although the iterator would be valid during the function call (the temporary vector would be destroyed after the call), `find()` returns a `std::ranges::dangling` object.

Again, the best way to deal with this issue is to declare a reference for the return value of `getData()`:

- Using a `const` lvalue reference:

  ```
  const auto& data = getData();                 // give return value a name to use it as an lvalue
  process(std::ranges::find(data, 42));         // passes a valid iterator to process()
  ```

- Using a *universal/forwarding reference*:

  ```
  auto&& data = getData();                      // give return value a name to use it as an lvalue
  process(std::ranges::find(data, 42));         // passes a valid iterator to process()
  ```

Remember that often, you need a name for the return value anyway to check whether the return value refers to an element and is not the `end()` of a range:

```
auto&& data = getData();                        // give return value a name to use it as an lvalue
auto pos = std::ranges::find(data, 42);         // yields a valid iterator
if (pos != data.end()) {                        // OK
  std::cout << *pos;                            // OK
}
```

6.2.2 Borrowed Ranges

Range types can claim that they are ***borrowed ranges***. This means that their iterators can still be used when the range itself no longer exists.

C++20 provides the concept `std::ranges::borrowed_range` for this. This concept is satisfied either if iterators of the range type never depend on the lifetime of their range ***or*** if the passed range object is an lvalue. This means that in a concrete situation, the concept checks whether iterators created for the range can be used after the range no longer exists.

All the standard containers and the views that refer to ranges passed as rvalues (temporary range objects) are not borrowed ranges because the iterators iterate over values stored inside them. For other views it depends. In those cases, we have two approaches that let views become borrowed ranges:

- The iterators store all information to iterate locally. For example:
  - `std::ranges::iota_view`, which generates an incrementing sequence of values. Here, the iterator stores the current value locally and does not refer to any other object.
  - `std::ranges::empty_view`, for which any iterator is always at the end so that it cannot iterate over element values at all.
- Iterators directly refer to the underlying range without using the view for which `begin()` and `end()` was called. For example, this is the case for:
  - `std::ranges::subrange`
  - `std::ranges::ref_view`
  - `std::span`
  - `std::string_view`

Note that borrowed iterators can still dangle when they refer to an underlying range (the latter category above) and the underlying range is no longer there.

As a result, we can catch ***some*** but not all possible runtime errors at compile time, which I can demonstrate with the various ways to try to find an element with the value 8 in various ranges (yes, we should usually check whether an end iterator was returned):

- All lvalues (objects with names) are borrowed ranges, which means that the returned iterator cannot be dangling as long as the iterator exists in the same scope or a sub-scope of the range.

```
std::vector coll{0, 8, 15};

auto pos0 = std::ranges::find(coll, 8);              // borrowed range
std::cout << *pos0;                                  // OK (undefined behavior if no 8)

auto pos1 = std::ranges::find(std::vector{8}, 8);    // yields dangling
std::cout << *pos1;                                  // compile-time ERROR
```

- For temporary views the situation depends. For example:

```
auto pos2 = std::ranges::find(std::views::single(8), 8);   // yields dangling
std::cout << *pos2;                                  // compile-time ERROR

auto pos3 = std::ranges::find(std::views::iota(8), 8);     // borrowed range
std::cout << *pos3;                            // OK (undefined behavior if no 8 found)

auto pos4 = std::ranges::find(std::views::empty<int>, 8);  // borrowed range
std::cout << *pos4;                            // undefined behavior as no 8 found
```

  For example, single view iterators refer to the value of their element in the view; therefore, single views are not borrowed ranges.

  On the other hand, iota view iterators hold copies of the element they refer to, which means that iota views are declared as borrowed ranges.

- For views that refer to another range (as a whole or to a sub-sequence of it), the situation is more complicated. If they can, they try to detect similar problems. For example, the adaptor `std::views::take()` also checks for prvalues:

```
auto pos5 = std::ranges::find(std::views::take(std::vector{0, 8, 15}, 2), 8);
                                               // compile-time ERROR
```

  Here, calling `take()` is already a compile-time error.

  However, if you use `counted()`, which only takes an iterator, it is the responsibility of the programmer to ensure that the iterator is valid:

```
auto pos6 = std::ranges::find(std::views::counted(std::vector{0, 8, 15}.begin(),
                                                  2), 8);
std::cout << *pos6;                            // runtime ERROR even if 8 found
```

  The views, which are created with `counted()` here, are by definition borrowed ranges, because they pass their internal references to their iterators. In other words: an iterator of a counted view does not need the view it belongs to. However, the iterator can still refer to a range that no longer exists (because its view referred to an object that no longer exists). The last line of the example with `pos6` demonstrates this situation. We still get undefined behavior even if the value that `find()` is looking for could be found in the temporary range.

If you implement a container or a view, you can signal that it is a borrowed range by specializing the variable template `std::ranges::enable_borrowed_range<>`.

6.3 Using Views

As introduced, views are lightweight ranges that can be used as building blocks to deal with the (modified) values of all or some elements of other ranges and views.

C++20 provides several standard views. They can be used to convert a range into a view or yield a view to a range/view where the elements are modified in various ways:

- Filter out elements
- Yield transformed values of elements
- Change the order of iterating over elements
- Split or merge ranges

In addition, there are some views that generate values themselves.

For almost every view type, there is a corresponding *customization point object* (usually a function object) that allows programmers to create a view by calling a function. Such a function object is called a ***range adaptor*** if the function creates a view from a passed range, or a ***range factory*** if the function creates a view without passing an existing range. Most of the time, these function objects have the name of the view without the `_view` suffix. However, some more general function objects might create different views depending on the passed arguments. These function objects are all defined in the special namespace `std::views`, which is an alias to the namespace `std::ranges::views`.

Table *Source views* lists the standard views of C++20 that create a view from an external resource or generate values themselves. These views especially can serve as the starting building blocks in a pipeline of views. You can also see which range adaptors or factories may create them (if there are any). If available, you should prefer using them rather than the raw view types.

If not specified differently, the adaptors and factories are available in the namespace `std::views` and the view types are available in the namespace `std::ranges`. `std::string_view` was already introduced with C++17. All other views were introduced with C++20 and usually end with `_view`. The only view types not ending with `_view` are `std::subrange` and `std::span`.

Table *Adapting views* lists the range adaptors and standard views of C++20 that process ranges and other views. They can serve as a building block anywhere in a pipeline of views, including at their beginning. Again, you should prefer to use the adaptors.

All of the views provide move (and optional copy) operations with constant complexity (the time these operations take does not depend on the number of elements).[2] The concept `std::ranges::view` checks the corresponding requirements.

The range factories/adaptors `all()`, `counted()`, and `common()` are described in a special section. Details of all view types and the other adaptors and factories are described in the chapter about *View Types in Detail*.

[2] The original C++20 standard also required views to have a default constructor and a destructor with constant complexity. However, these requirements were removed later with `http://wg21.link/P2325R3` and `http://wg21.link/p2415r2`.

Adaptor/Factory	**Type**	**Effect**
`all(`*rg*`)`	Varies:	Yields range *rg* as a view
	type of *rg*	- Yields *rg* if it is already a view
	`ref_view`	- Yields a `ref_view` if *rg* is an lvalue
	`owning_view`	- Yields an `owning_view` if *rg* is an rvalue
`counted(`*beg*`,`*sz*`)`	Varies:	Yields a view from a begin iterator and a count
	`std::span`	- Yields a span if *rg* is contiguous and common
	`subrange`	- Yields a `subrange` otherwise (if valid)
`iota(`*val*`)`	`iota_view`	Yields an endless view with an incrementing sequence of values starting with *val*
`iota(`*val*`,` *endVal*`)`	`iota_view`	Yields a view with an incrementing sequence of values from *val* up to (but not including) *endVal*
`single(`*val*`)`	`single_view`	Yields a view with *val* as the only element
`empty<`*T*`>`	`empty_view`	Yields an empty view of elements of type *T*
–	`basic_istream_view`	Yields a view that reads *T*s from input stream *s*
`istream<`*T*`>(`*s*`)`	`istream_view`	Yields a view that reads *T*s from `char` stream *s*
–	`wistream_view`	Yields a view that reads *T*s from `wchar_t` stream *s*
–	`std::basic_string_view`	Yields a read-only view to a character array
–	`std::span`	Yields a view to elements in contiguous memory
–	`subrange`	Yields a view for a begin iterator and a sentinel

Table 6.3. Source views

Adaptor	**Type**	**Effect**
`take(`*num*`)`	Varies	The first (up to) *num* elements
`take_while(`*pred*`)`	`take_while_view`	All leading elements that match a predicate
`drop(`*num*`)`	Varies	All except the first *num* elements
`drop_while(`*pred*`)`	`drop_while_view`	All except leading elements that match a predicate
`filter(`*pred*`)`	`filter_view`	All elements that match a predicate
`transform(`*func*`)`	`transform_view`	The transformed values of all elements
`elements<`*idx*`>`	`elements_view`	The *idx*th member/attribute of all elements
`keys`	`elements_view`	The first member of all elements
`values`	`elements_view`	The second member of all elements
`reverse`	Varies	All elements in reverse order
`join`	`join_view`	All elements of a range of multiple ranges
`split(`*sep*`)`	`split_view`	All elements of a range split into multiple ranges
`lazy_split(`*sep*`)`	`lazy_split_view`	All elements of an input or `const` range split into multiple ranges
`common`	Varies	All elements with same type for iterator and sentinel

Table 6.4. Adapting views

6.3.1 Views on Ranges

Containers and strings are not views. This is because they are not lightweight enough: they provide no cheap copy constructors because they have to copy the elements.

However, you can easily use containers as views:

- You can explicitly convert a container to a view by passing it to the range adaptor `std::views::all()`.
- You can explicitly convert elements of a container to a view by passing a begin iterator and an end (sentinel) or a size to a `std::ranges::subrange` or `std::views::counted()`.
- You can implicitly convert a container to a view by passing it to one of the adapting views. These views usually take a container by converting it implicitly to a view.

Usually, the latter option is and should be used. There are multiple ways of implementing this option. For example, you have the following options for passing a range *coll* to a take view:

- You can pass the range as a parameter to the constructor of the view:
  ```
  std::ranges::take_view first4{coll, 4};
  ```
- You can pass the range as a parameter to the corresponding adaptor:
  ```
  auto first4 = std::views::take(coll, 4);
  ```
- You can pipe the range into the corresponding adaptor:
  ```
  auto first4 = coll | std::views::take(4);
  ```

In any case, the view `first4` iterates over only the first four elements of *coll* (or fewer if there are not enough elements). However, what happens exactly here depends on what *coll* is:

- If *coll* is already a view, `take()` just takes the view as it is.
- If *coll* is a container, `take()` uses a view to the container that is automatically created with the adaptor `std::views::all()`. This adaptor yields a `ref_view` that refers to all elements of the container if the container is passed by name (as an lvalue).
- If an rvalue is passed (a temporary range such as a container returned by a function or a container marked with `std::move()`), the range is moved into an `owning_view`, which then holds a range of the passed type with all moved elements directly.[3]

For example:

```
std::vector<std::string> coll{"just", "some", "strings", "to", "deal", "with"};

auto v1 = std::views::take(coll, 4);              // iterates over a ref_view to coll

auto v2 = std::views::take(std::move(coll), 4);   // iterates over an owning_view
                                                  // to a local vector<string>

auto v3 = std::views::take(v1, 2);                // iterates over v1
```

[3] The support of temporary objects (rvalues) for views was added after C++20 was published with http://wg21.link/p2415r2.

In all cases, `std::views::take()` creates a new take view that finally iterates over the values initialized in `coll`. However, the resulting types and exact behavior differ as follows:

- `v1` is a `take_view<ref_view<vector<string>>>`.
  Because we pass the container `coll` as lvalue (named object), the take view iterates over a ref view to the container.
- `v2` is a `take_view<owning_view<vector<string>>>`.
  Because we pass `coll` as an rvalue (temporary object or object marked with `std::move()`), the take view iterates over an owning view that holds its own vector of strings move-initialized with the passed collection.
- `v3` is a `take_view<take_view<ref_view<vector<string>>>>`.
  Because we pass the view `v1`, the take view iterates over this view. The effect is that we finally iterate over `coll` (for which the elements were moved away with the second statement, so do not do that after the second statement).

Internally, the initialization uses the deduction guides and the type utility `std::views::all_t<>`, which is explained later in detail.

Note that this behavior allows a range-based `for` loop to iterate on temporary ranges:

```
for (const auto& elem : getColl() | std::views::take(5)) {
  std::cout << "- " << elem << '\n';
}

for (const auto& elem : getColl() | std::views::take(5) | std::views::drop(2)) {
  std::cout << "- " << elem << '\n';
}
```

This is remarkable, because in general, it is a fatal runtime error to use a reference to a temporary as a collection a range-based `for` loop iterates over (a bug that the C++ standards committee has not been willing to fix for years now; see `http://wg21.link/p2012`). Because passing a temporary range object (rvalue) moves the range into an `owning_view`, the view does not refer to an external container and therefore there is no runtime error.

6.3.2 Lazy Evaluation

It is important to understand when exactly views are processed. Views do not start processing when they are defined; instead, they run on demand:

- If we need the next element of a view, we compute which one it is by performing the necessary iteration(s).
- If we need the value of an element of a view, we compute its value by performing the defined transformation(s).

Consider the following program:

ranges/filttrans.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

```

```
int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
                 | vws::filter([] (int i) {
                                 std::cout << " filter " << i << '\n';
                                 return i % 3 == 0;
                               })
                 | vws::transform([] (int i) {
                                    std::cout << "  trans " << i << '\n';
                                    return -i;
                                  });

  // and use it:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    std::cout << "val: " << val << "\n\n";
  }
}
```

We define a view `vColl` that only filters and transforms the elements of the range `coll`:

- By using `std::views::filter()`, we process only those elements that are a multiple of 3.
- By using `std::views::transform()`, we negate each value.

The program has the following output:

```
*** coll | filter | transform:
 filter 8
 filter 15
  trans 15
val: -15

 filter 7
 filter 0
  trans 0
val: 0

 filter 9
  trans 9
val: -9
```

First, note that neither `filter()` nor `transform()` are called when or after we define the view `vColl`. The processing starts when we *use* the view (here: iterating over `vColl`). This means that views use ***lazy***

evaluation. A view is just the description of a processing. The processing is performed when we need the next element or value.

Let us assume that we do the iteration over `vColl` more manually, calling `begin()` and `++` to get the next value and `*` to get its value:

```
std::cout << "pos = vColl.begin():\n";
auto pos = vColl.begin();
std::cout << "*pos:\n";
auto val = *pos;
std::cout << "val: " << val << "\n\n";

std::cout << "++pos:\n";
++pos;
std::cout << "*pos:\n";
val = *pos;
std::cout << "val: " << val << "\n\n";
```

For this code, we get the following output:

```
pos = vColl.begin():
 filter 8
 filter 15
*pos:
  trans 15
val: -15

++pos:
 filter 7
 filter 0
*pos:
  trans 0
val: 0
```

Let us look at what happens step by step:

- When we call `begin()`, the following happens:
  - The request to get the first element of `vColl` is passed to the transform view, which passes it to the filter view, which passes it to `coll`, which yields an iterator to the first element `8`.
  - The filter view looks at the value of the first element and rejects it. As a consequence, it asks `coll` for the next element by calling `++` there. The filter gets the position of the second element `15` and passes its position to the transform view.
  - As a result, `pos` is initialized as an iterator with the position to the second element.
- When calling `*pos`, the following happens:
  - Because we need the value, the transform view is now called for the current element and yields its negated value.
  - As a result, `val` is initialized with the negated value of the current element.

- When calling `++pos`, the same happens again:
  - The request to get the next element is passed to the filter, which passes the request to `coll` until an element fits (or we are at the end of `coll`).
  - As a result, `pos` gets the position of the fourth element.
- By calling `*pos` again, we perform the transformation and yield the next value for the loop.

This iteration continues until the range or one of the views signals that we are at the end.

This ***pull model*** has a big benefit: we do not process elements that we never need. For example, assume that we use the view to find the first resulting value that is `0`:

```
std::ranges::find(vColl, 0);
```

The output would then be only:

```
filter 8
filter 15
 trans 15
filter 7
filter 0
 trans 0
```

Another benefit of the pull model is that sequences or pipelines of views can even operate on infinite ranges. We do not compute an unlimited number of values not knowing how many are used; we compute as many values as requested by the user of the view.

6.3.3 Caching in Views

Assume that we want to iterate over a view multiple times. It seems a waste of performance if we compute the first valid element again and again. And indeed, views that might skip leading elements ***cache*** the `begin()` once it is called.

Let us change the program above so that it iterates twice over the elements of the view `vColl`:

ranges/filttrans2.cpp

```cpp
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
               | vws::filter([] (int i) {
                              std::cout << " filter " << i << '\n';
                              return i % 3 == 0;
                            })
```

```
                 | vws::transform([] (int i) {
                                  std::cout << "  trans " << i << '\n';
                                  return -i;
                                });

  // and use it:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    ...
  }
  std::cout << "--------------------\n";

  // and use it again:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    std::cout << "val: " << val << "\n\n";
  }
}
```

The program has the following output:

```
*** coll | filter | transform:
 filter 8
 filter 15
  trans 15
 filter 7
 filter 0
  trans 0
 filter 9
  trans 9
--------------------
*** coll | filter | transform:
  trans 15
val: -15

 filter 7
 filter 0
  trans 0
val: 0

 filter 9
  trans 9
val: -9
```

As you can see, when used for the second time, calling `vColl.begin()` no longer tries to find the first element because it was cached with the first iteration over the elements of the filter.

Note that this caching of `begin()` has good and bad and maybe unexpected consequences. First, it is better to initialize a caching view once and use it twice:

```
// better:
auto v1 = coll | std::views::drop(5);
check(v1);
process(v1);
```

than to initialize and use it twice:

```
// worse:
check(coll | std::views::drop(5));
process(coll | std::views::drop(5));
```

In addition, modifying leading elements of ranges (changing their value or inserting/deleting elements) may invalidate views ***if and only if*** `begin()` has already been called before the modification.

That means:

- If we do not call `begin()` before a modification, the view is usually valid and works fine when we use it later:

  ```
  std::list coll{1, 2, 3, 4, 5};
  auto v = coll | std::views::drop(2);
  coll.push_front(0);   // coll is now: 0 1 2 3 4 5
  print(v);             // initializes begin() with 2 and prints: 2 3 4 5
  ```

- However, if we ***do*** call `begin()` before the modification (e.g., by printing the elements), we can easily get wrong elements. For example:

  ```
  std::list coll{1, 2, 3, 4, 5};
  auto v = coll | std::views::drop(2);
  print(v);             // init begin() with 3
  coll.push_front(0);   // coll is now: 0 1 2 3 4 5
  print(v);             // begin() is still 3, so prints: 3 4 5
  ```

  Here, `begin()` is cached as an iterator so that the view no longer operates on all elements of the underlying range if new elements are added to or removed from the range.

  However, we might also get invalid values. For example:

  ```
  std::vector vec{1, 2, 3, 4};

  auto biggerThan2 = [](auto v){ return v > 2; };
  auto vVec = vec | std::views::filter(biggerThan2);

  print(vVec);     // OK: 3 4

  ++vec[1];
  vec[2] = 0;      // vec becomes 1 3 0 4

  print(vVec);     // OOPS: 0 4;
  ```

Note that this means that an iteration, even if it only reads, may count as a write access. Therefore, **iterating over the elements of a view might invalidate a later use** if its referred range has been modified in the meantime.

The effect depends on when and how caching is done. See the remarks about caching views in their specific sections:

- Filter views, which cache `begin()` as an iterator or offset
- Drop views, which cache `begin()` as an iterator or offset
- Drop-while views, which cache `begin()` as an iterator or offset
- Reverse views, which cache `begin()` as an iterator or offset

Here, you see again that C++ cares about performance:

- Caching at initialization time would have unnecessary performance cost if we never iterate over the elements of a view at all.
- Not caching at all would have unnecessary performance cost if we iterate a second or more times over the elements of the view (applying a reverse view over a drop-while view might even have quadratic complexity in some cases).

However, due to caching, using a view not ad hoc can have pretty surprising consequences. Care must be taken when modifying ranges used by views.

As another consequence, caching might require that a view cannot be `const` while iterating over its elements. The consequences are even more severe and are discussed later.

6.3.4 Performance Issues with Filters

The pull model also has its drawbacks. To demonstrate that, let us change the order of the two views involved (above) so that we first call `transform()` and then `filter()`:

ranges/transfilt.cpp

```cpp
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
               | vws::transform([] (int i) {
                                  std::cout << " trans: " << i << '\n';
                                  return -i;
                                })
               | vws::filter([] (int i) {
                               std::cout << "  filt: " << i << '\n';
```

```
                                        return i % 3 == 0;
                                      });

  // and use it:
  std::cout << "*** coll | transform | filter:\n";
  for (int val : vColl) {
    std::cout << "val: " << val << "\n\n";
  }
}
```

Now, the output of the program is as follows:

```
*** coll | transform | filter:
 trans: 8
  filt: -8
 trans: 15
  filt: -15
 trans: 15
val: -15

 trans: 7
  filt: -7
 trans: 0
  filt: 0
 trans: 0
val: 0

 trans: 9
  filt: -9
 trans: 9
val: -9
```

We have additional calls of the transform view:

- We now call `transform()` for each element.
- For elements that pass the filter, we even perform the transformation twice.

Calling the transformation for each element is necessary because the filter view comes afterwards and now looks at the transformed values. In this case, the negation does not impact the filter; therefore, putting it to the front looks better.

However, why do we call the transformations for elements that pass the filter twice? The reason lies in the nature of a pipeline that uses the pull model and the fact that we use iterators.

- First, the filter needs the transformed value to check it; therefore, the leading transformation has to be performed before the filter can use the resulting value.
- Remember that ranges and views iterate in two steps over the elements of a range: they first compute the position/iterator (with `begin()` and `++`), and then as a separate step, use `*` to get the value. This means that for each filter, we have to perform all previous transformations once to be able *check* the value of the

element. However, if `true`, the filter delivers only the position of the element, not its value. Therefore, when a user of the filter needs the value, it has to perform the transformations once again.

In fact, each filter adds one more call of all leading transformations on each element that pass the filter and for which the value is used afterwards.

Given the following pipeline of transformations `t1`, `t2`, `t3` and filters `f1`, `f2`:

```
t1 | t2 | f1 | t3 | f2
```

we have the following behavior:

- For elements where `f1` yields `false`, we call:
  `t1 t2 f1`
- For elements where `f1` yields `true` but `f2` yields `false`, we call:
  `t1 t2 f1 t1 t2 t3 f2`
- For elements where `f1` and `f2` yield `true`, we call:
  `t1 t2 f1 t1 t2 t3 f2 t1 t2 t3`

See *ranges/viewscalls.cpp* for a complete example.

If this makes you worry about the performance of pipelines, take the following into account: yes, you should avoid expensive transformations before you use a filter. However, remember that all views provide functionality that is usually optimized in such a way that only the expressions inside the transformations and filters remain. In trivial cases such as the one here, where a filter checks for a multiple of 3 and the transformation negates, the resulting difference in behavior is effectively like calling

```
if (-x % 3 == 0) return -x;      // first transformation then filter
```

instead of

```
if (x % 3 == 0) return -x;       // first filter then transformation
```

For more details about filters, see the section about the filter view.

6.4 Views on Ranges That Are Destroyed or Modified

Views usually have reference semantics. Often, they refer to ranges that exist outside themselves. This means that care must be taken because you can only use views as long as the underlying ranges exist and references to them stored in the views or its iterators are valid.

6.4.1 Lifetime Dependencies Between Views and Their Ranges

All views that operate on a range passed as an lvalue (as a first constructor argument or using a pipe) store a reference to the passed range internally.

This means that the underlying range still has to exist when the view is used. Code like this causes undefined behavior:

```
auto getValues()
{
  std::vector coll{1, 2, 3, 4, 5};
  ...
  return coll | std::views::drop(2);   // ERROR: return reference to local range
}
```

What we return here is a drop view by value. However, internally, it refers to `coll`, which is destroyed with the end of `getValues()`.

This code is as bad as returning a reference or pointer to a local object. It might work accidentally or cause a fatal runtime error. Unfortunately, compilers do not warn about this (yet).

Using views on range objects that are rvalues works fine. You can return a view to a temporary range object:

```
auto getValues()
{
  ...
  return std::vector{1, 2, 3, 4, 5} | std::views::drop(2);  // OK
}
```

Or you can mark the underlying range with `std::move()`:

```
auto getValues()
{
  std::vector coll{1, 2, 3, 4, 5};
  ...
  return std::move(coll) | std::views::drop(2);             // OK
}
```

6.4.2 Views with Write Access

Views should never modify a passed argument or call a non-`const` operation for it. That way, views and their copies have the same behavior for the same input.

For views that use helper functions to check or transform values, this means that these helper functions should never modify the elements. Ideally, they should take the values either by value or by `const` reference. If you modify the argument passed as a non-`const` reference, you have undefined behavior:

```
coll | std::views::transform([] (auto& val) {  // better declare val as const&
                                ++val;         // ERROR: undefined behavior
                              })

coll | std::views::drop([] (auto& val) {       // better declare val as const&
                          return ++val > 0;    // ERROR: undefined behavior
                        })
```

Note that compilers cannot check whether a helper function or predicate modifies the passed value. The views require that the passed functions or predicates are `std::regular_invocable` (which is implicitly required by `std::predicate`). However, not modifying the value is a semantic constraint and this cannot always be checked at compile time. Therefore, it is up to you to make it right.

However, there ***is*** support for using views to restrict the subset of elements you want to modify. For example:

```
// assign 0 to all but the first five elements of coll:
for (auto& elem : coll | vws::drop(5)) {
  elem = 0;
}
```

6.4.3 Views on Ranges That Change

Views that cache `begin()` may run into significant problems if they are not used ad hoc and the underlying ranges change (note that I am writing about problems that occur when both the range and calling `begin()` for it are still valid).

Consider the following program:

ranges/viewslazy.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << ' ' << elem;
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5};
  std::list lst{1, 2, 3, 4, 5};

  auto over2 = [](auto v) { return v > 2; };
  auto over2vec = vec | std::views::filter(over2);
  auto over2lst = lst | std::views::filter(over2);

  std::cout << "containers and elements over 2:\n";
  print(vec);                 // OK:  1 2 3 4 5
  print(lst);                 // OK:  1 2 3 4 5
  print(over2vec);            // OK:  3 4 5
  print(over2lst);            // OK:  3 4 5

  // modify underlying ranges:
  vec.insert(vec.begin(), {9, 0, -1});
  lst.insert(lst.begin(), {9, 0, -1});

  std::cout << "containers and elements over 2:\n";
  print(vec);                 // vec now: 9 0 -1 1 2 3 4 5
  print(lst);                 // lst now: 9 0 -1 1 2 3 4 5
  print(over2vec);            // OOPS:  -1 3 4 5
  print(over2lst);            // OOPS:  3 4 5
  // copying might eliminate caching:
  auto over2vec2 = over2vec;
  auto over2lst2 = over2lst;
  std::cout << "elements over 2 after copying the view:\n";
  print(over2vec2);          // OOPS: -1 3 4 5
  print(over2lst2);          // OK: 9 3 4 5
}
```

It has the following output:

```
containers and elements over 2:
 1 2 3 4 5
 1 2 3 4 5
 3 4 5
 3 4 5
containers and elements over 2:
 9 0 -1 1 2 3 4 5
 9 0 -1 1 2 3 4 5
 -1 3 4 5
 3 4 5
elements over 2 after copying the view:
 -1 3 4 5
 9 3 4 5
```

The problem is the caching of the filter view. With the first iteration, both views cache the beginning of the view. Note that this happens in different ways:

- For a random-access range like a vector, the view caches the offset to the first element. That way, a reallocation that invalidates `begin()` does not cause undefined behavior when the view is used again.
- For other ranges like a list, the view really caches the result of calling `begin()`. This can be a severe problem if the cached element is removed meaning that the cached `begin()` is no longer valid. However, it can also create confusion if new elements are inserted ahead.

The overall effect of caching is that further modifications of the underlying range might invalidate the view in various ways:

- A cached `begin()` may no longer be valid.
- A cached offset may be after the end of the underlying range.
- New elements that fit the predicate might not be used by later iterations.
- New elements that do not fit might be used by later iterations.

6.4.4 Copying Views Might Change Behavior

Finally, note that the previous example demonstrates that copying a view may sometimes invalidate the cache:

- The views that have cached `begin()` have this output:

  ```
  -1 3 4 5
  3 4 5
  ```

- The copies of these views have a different output:

  ```
  -1 3 4 5
  9 3 4 5
  ```

The cached offset for the view to the vector is still used after copying, while the cached `begin()` for the view to the list (`begin()`) was removed.

This means that caching has the additional consequence that a copy of a view might not have the same state as its source. For this reason, you should think twice before making a copy of a view (although one design goal was to make copying cheap to pass them by value).

If there is one consequence of this behavior, it is the following: **use views ad hoc (immediately after you define them).**

This is a little bit sad, because the lazy evaluation of views, in principle, would allow some incredible use cases, which cannot be implemented in practice because the behavior of code is hard to predict if views are not used ad hoc and underlying ranges might change.

Write Access with Filter Views

When using filter views, there are important additional restrictions on write access:

- First, as described before, because filter views cache, modified elements might pass the filter even though they should not.
- In addition, you have to ensure that the modified value still fulfills the predicate passed to the filter. Otherwise, you get undefined behavior (which sometimes yields the correct results, though). See the description of the filter view for details and an example.

6.5 Views and `const`

When using views (and the ranges library in general), there are a few things that are surprising or even broken regarding constness:

- For some views, it is not possible to iterate over the elements when the view is `const`.
- Views remove the propagation of constness to elements.
- Functions like `cbegin()` and `cend()`, which have the goal of ensuring that elements are `const` while iterating over them, are either not provided or are actually broken.

There are more or less good reasons for these problems. Some of them have to do with the nature of views and some basic design decisions that were made. I consider this all as a **severe design mistake**. However, others in the C++ standards committee have different opinions.

There are fixes under way that at least fix the broken constness of `cbegin()` and `cend()`. Unfortunately, the C++ standards committee decided not to apply them to C++20. These will come (and change behavior) with C++23. See `http://wg21.link/p2278r4` for details.

6.5.1 Generic Code for Both Containers and Views

It is surprisingly complicated to implement a generic function that can iterate over the elements of every type of containers and views with good performance. For example, declaring a function that shall print all elements as follows does ***not*** always work:

```
template<typename T>
void print(const T& coll);        // OOPS: might not work for some views
```

Consider the following concrete example:

ranges/printconst.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(const auto& rg)
{
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::list   lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

  print(vec | std::views::take(3));                      // OK
  print(vec | std::views::drop(3));                      // OK

  print(lst | std::views::take(3));                      // OK
  print(lst | std::views::drop(3));                      // ERROR
  for (const auto& elem : lst | std::views::drop(3)) {   // OK
    std::cout << elem << ' ';
  }
  std::cout << '\n';

  auto isEven = [] (const auto& val) {
    return val % 2 == 0;
  };
  print(vec | std::views::filter(isEven));               // ERROR
}
```

First, we declare a very trivial generic function `print()`, which prints all elements of a range passed as a `const` reference:

```
void print(const auto& rg)
{
  ...
}
```

Then, we call `print()` for a couple of views and run into surprising behavior:

- Passing take views and a drop view to a vector works fine:

  ```
  print(vec | std::views::take(3));                   // OK
  print(vec | std::views::drop(3));                   // OK

  print(lst | std::views::take(3));                   // OK
  ```

- Passing a drop view to a list does not compile:

  ```
  print(vec | std::views::take(3));                   // OK
  print(vec | std::views::drop(3));                   // OK

  print(lst | std::views::take(3));                   // OK
  print(lst | std::views::drop(3));                   // ERROR
  ```

- However, iterating over the drop view directly works fine:

  ```
  for (const auto& elem : lst | std::views::drop(3)) {   // OK
    std::cout << elem << ' ';
  }
  ```

This does not mean that vectors always work fine. If, for example, we pass a filter view, we get an error for all ranges:

```
print(vec | std::views::filter(isEven));              // ERROR
```

`const&` Does Not Work for All Views

The reason for this very strange and unexpected behavior is that some views do not always support iterating over elements when the view is `const`. It is a consequence of the fact that iterating over the elements of these views sometimes needs the ability to modify the state of the view (e.g., due to caching). For some views (such as the filter view), it never works; for some views (such as the drop view), it only works sometimes.

In fact, you cannot iterate over the elements of the following standard views if they are declared with `const`:[4]

- `const` views that you can ***never*** iterate over:
  - Filter view
  - Drop-while view
  - Split view
  - IStream view

[4] Thanks to Hannes Hauswedell for pointing out missing parts of this list.

- `const` views that you can only ***sometimes*** iterate over:
  - Drop view, if it refers to a range that has no random access or no `size()`
  - Reverse view, if it refers to a range that has different types for the begin iterator and the sentinel (end iterator)
  - Join view, if it refers to a range that generates values instead of references
  - All views that refer to other ranges if the referred ranges themselves are not `const` iterable

For these views, `begin()` and `end()` as `const` member functions are provided only conditionally or are not provided at all. For example, for drop views, `begin()` for `const` objects is provided only if the passed ranges fulfill the requirement of a random-access range and sized range:[5]

```
namespace std::ranges {
  template<view V>
  class drop_view : public view_interface<drop_view<V>> {
    public:
      ...
      constexpr auto begin() const requires random_access_range<const V>
                                       && sized_range<const V>;
      ...
  };
}
```

Here, we use the new feature to constrain member functions.

This means that you cannot provide a generic function that can deal with the elements of all ranges and views if you declare the parameter to be a `const` reference.

```
void print(const auto& coll);      // not callable for all views

template<typename T>
void foo(const T& coll);           // not callable for all views
```

It does not matter whether you have constraints for the type of the range parameter:

```
void print(const std::ranges::input_range auto& coll);      // not callable for all views

template<std::ranges::random_access_range T>
void foo(const T& coll);                                     // not callable for all views
```

[5] You might be surprised by the fact that the template parameter `V` is constrained to be a view. However, it may be a ref view implicitly created from a passed container that is not a view.

Non-const && Does Work for All Views

To also support these views in generic code, you should declare the range parameters as *universal references* (also called *forwarding references*). These are references that can refer to all expressions while retaining the fact that the referred object is not const.[6] For example:

```
void print(std::ranges::input_range auto&& coll);      // can in principle pass all views

template<std::ranges::random_access_range T>
void foo(T&& coll);                                      // can in principle pass all views
```

Thus, the following program works fine:

ranges/printranges.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& rg)
{
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::list   lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

  print(vec | std::views::take(3));                        // OK
  print(vec | std::views::drop(3));                        // OK

  print(lst | std::views::take(3));                        // OK
  print(lst | std::views::drop(3));                        // OK
  for (const auto& elem : lst | std::views::drop(3)) {     // OK
    std::cout << elem << ' ';
  }
  std::cout << '\n';
```

[6] In the C++ standard, universal references are called ***forwarding references***, which is unfortunate because they do not really forward until `std::forward<>()` is called for them, and perfect forwarding is not always the reason to use these references (as this example demonstrates).

```
  auto isEven = [] (const auto& val) {
    return val % 2 == 0;
  };
  print(vec | std::views::filter(isEven));                // OK
}
```

For the same reason, the generic `maxValue()` function introduced before does not always work:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  ...   // ERROR when iterating over the elements of a filter view
}

// max value of a filtered range:
auto odd = [] (auto val) {
             return val % 2 != 0;
           };
std::cout << maxValue(arr | std::views::filter(odd)) << '\n';   // ERROR
```

As written, it is better to implement the generic `maxValue()` function as follows:

ranges/maxvalue2.hpp

```
#include <ranges>

template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
}
```

Here, we declare parameter `rg` as a universal/forwarding reference:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

That way, we ensure that the passed views are not becoming `const`, which means that we can also pass a filter view now:

ranges/maxvalue2.cpp

```
#include "maxvalue2.hpp"
#include <iostream>
#include <algorithm>

int main()
{
  int arr[] = {0, 8, 15, 42, 7};

  // max value of a filtered range:
  auto odd = [] (auto val) {                          // predicate for odd values
               return val % 2 != 0;
             };
  std::cout << maxValue(arr | std::views::filter(odd)) << '\n';   // OK
}
```

The output of the program is:

```
15
```

Are you wondering now how to declare a generic function that can be called for all `const` and non-`const` ranges and views and guarantees that the elements are not modified? Well, again, there is an important lesson to learn here:

> **Since C++20, there is no longer a way to *declare* a generic function to takes all standard collection types (`const` and non-`const` containers and views) with the guarantee that the elements are not modified.**

All you can do is to take the range/view and ensure in the ***body*** of your function that elements are not modified. However, it might be surprisingly complicated to do that because as we will now see:

- Declaring views as `const` does not necessarily make the elements `const`.
- Views do not have `const_iterator` members.
- Views do not provide `cbegin()` and `cend()` members (yet).
- `std::ranges::cbegin()` and `std::cbegin()` are broken for views.
- Declaring a view element as `const` might have no effect.

Use `const&` for Concurrent Iterations

Regarding the advice to use universal/forwarding references, there is an important restriction: you should not use them when you iterate over views concurrently.

Consider the following example:

```
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8};

auto v = lst | std::views::drop(2);

// while another thread prints the elements of the view:
std::jthread printThread{[&] {
                           for (const auto& elem : v) {
                             std::cout << elem << '\n';
                           }
                         }};

// this thread computes the sum of the element values:
auto sum = std::accumulate(v.begin(), v.end(),           // fatal runtime ERROR
                           0L);
```

By using a std::jthread, we start a thread that iterates over the elements of the view v to print them. At the same time, we iterate over v to compute the sum of the element values. For standard containers, parallel iterations that only read are safe (there is a guarantee for containers, that calling begin() counts as read access). However, this guarantee does not apply to standard views. Because we might call begin() for view v concurrently, this code causes undefined behavior (a possible *data race*).

Functions that perform concurrent iterations that only read should use const view to convert possible runtime errors into compile-time errors:

```
const auto v = lst | std::views::drop(2);

std::jthread printThread{[&] {
                           for (const auto& elem : v) {  // compile-time ERROR
                             std::cout << elem << '\n';
                           }
                         }};

auto sum = std::accumulate(v.begin(), v.end(),           // compile-time ERROR
                           0L);
```

Overloading for Views

You might claim that you can simply overload a generic function for containers and views. For views, it might be enough to just add an overload that constrains the function for views and takes the parameter by value:

```
void print(const auto& rg);                  // for containers
void print(std::ranges::view auto rg)        // for views
```

However, the rules of overload resolution can become tricky when you once pass by value and once by reference. In this case, it causes ambiguities:

```
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3));            // ERROR: ambiguous
```

It also does not help to use a concept for the views that subsumes a concept used for the general case:

```
void print(const std::ranges::range auto& rg);   // for containers
void print(std::ranges::view auto rg)            // for views

std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3));                // ERROR: ambiguous
```

Instead, you have to state that the `const&` overload is not for views:

```
template<std::ranges::input_range T>             // for containers
requires (!std::ranges::view<T>)                 // and not for views
void print(const T& rg);

void print(std::ranges::view auto rg)            // for views

std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3));                // OK
```

However, remember that copying a view might create a view with different state and behavior than the source view. For this reason, it is anyway questionable whether it is good to pass all views by value.

6.5.2 Views May Remove the Propagation of const

Containers have *deep constness*. Because they have value semantics and own their elements, they propagate any constness to their elements. When a containers is `const`, its elements are `const`. As a consequence, the following code fails to compile:

```
template<typename T>
void modifyConstRange(const T& range)
{
  range.front() += 1;                  // modify an element of a const range
}

std::array<int, 10> coll{};            // array of 10 ints with value 0
...
modifyConstRange(coll);                // compile-time ERROR
```

This compile-time error is helpful because it helps to detect code that would otherwise be broken at runtime. For example, if you accidentally use the assignment operator instead of a comparison, the code does not compile:

```
template<typename T>
void modifyConstRange(const T& range)
{
  if (range[0] = 0) {                  // OOPS: should be ==
    ...
  }
}
```

```
std::array<int, 10> coll{};               // array of 10 ints with value 0
...
modifyConstRange(coll);                   // compile-time ERROR (thank goodness)
```

Knowing that elements cannot be modified also helps for optimizations (such as avoiding the need to make backups and check for changes) or to be sure that element access cannot cause undefined behavior due to data races when multiple threads are used.

With views the situation is more complicated. Views on ranges passed as lvalues have reference semantics. They ***refer*** to elements stored somewhere else. By design, these views do ***not*** propagate constness to their elements. They have ***shallow constness***:

```
std::array<int, 10> coll{};               // array of 10 ints with value 0
...
std::ranges::take_view v{coll, 5};        // view to first five elements of coll
modifyConstRange(v);                      // OK, modifies the first element of coll
```

That applies to almost all views on lvalues regardless of how they are created:

```
modifyConstRange(coll | std::views::drop(5));     // OK, elements of coll are not const
modifyConstRange(coll | std::views::take(5));     // OK, elements of coll are not const
```

As a consequence, you can use the range adaptor `std::views::all()` to pass a container to a generic function that takes a range as a `const` reference:

```
modifyConstRange(coll);                           // compile-time ERROR: elements are const
modifyConstRange(std::views::all(coll));          // OK, elements of coll are not const
```

Note that views on rvalues (if they are valid) usually still propagate constness:

```
readConstRange(getColl() | std::views::drop(5));    // compile-time ERROR (good)
readConstRange(getColl() | std::views::take(5));    // compile-time ERROR (good)
```

Also, note that it is not possible to modify the elements of a `const` container using a view:

```
const std::array<int, 10> coll{};         // array of 10 ints with value 0
...
auto v = std::views::take(coll, 5);       // view to first five elements of coll
modifyConstRange(v);                      // compile-time ERROR (good)
```

Therefore, you can ensure that elements of a container cannot be modified by a function that takes a view by making the container `const` ***before*** the view refers to it:

```
std::array<int, 10> coll{};                          // array of 10 ints with value 0
...
std::ranges::take_view v{std::as_const(coll), 5};    // view to five elems of const coll
modifyConstRange(v);                                               // compile-time ERROR
modifyConstRange(std::as_const(coll) | std::views::take(5)); // compile-time ERROR
```

The function `std::as_const()` has been provided since C++17 in the header file `<utility>`. However, calling `std::as_const()` *after* the view was created has no effect (except that for a few views, you are no longer able to iterate over the elements).

C++23 will introduce a helper view `std::ranges::as_const_view` with the range adaptor `std::views::as_const()` so that you can simply make the elements of a view `const` as follows (see http://wg21.link/p2278r4):

```
std::views::as_const(v);        // make elements of view v const
```

However, note again that you must not forget the namespace `views` here:

```
std::as_const(v);               // OOPS: no effect on views
```

The functions `as_const()` in the namespace `std` and in the namespace `std::ranges` both make something `const` but the former makes the object `const` while the latter makes the elements `const`.[7]

You might wonder why views with reference semantics were not designed in a way that they propagate constness. One argument was that they internally use pointers and therefore should behave like pointers. However, then it is strange that views on rvalues do not work that way. Another argument was that the constness is worth almost nothing because it can easily be removed by copying the view to a non-`const` view:

```
void foo(const auto& rg)      // if rg would propagate constness for views on lvalues
{
  auto rg2 = rg;              // rg2 would no longer propagate constness
  ...
}
```

The initialization would act kind of a `const_cast<>`. However, if this is generic code for containers programmers are already used ***not*** to copy a passed collection, because it is expensive to do so. Therefore, we now have another good reason not to copy a passed range. And the effect of declaring ranges (containers and views) as `const` would be way more consistent. However, the design decision is done now and as a programmer you have to deal with it.

6.5.3 Bringing Back Deep Constness to Views

As we have learned, using `const` for ranges that are views is a problem for two reasons:

- `const` may disable iteration over the elements
- `const` might not be propagated to the elements of a view

The obvious question then is how to ensure that code cannot modify the elements of a range when using a view.

Making the Elements `const` When Using Them

The only easy way to ensure that elements of a view cannot be modified is to force constness on element access:

- Either by using `const` in the range-based `for` loop:

  ```
  for (const auto& elem : rg) {
    ...
  }
  ```

[7] The members of the C++ standards committee sometimes decide in favor of features that look the same even though details differ and may become a source of trouble.

- Or by passing elements converted to `const`:

  ```
  for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
    elemfunc(std::as_const(*pos));
  }
  ```

However, here is the bad news: the designers of the views library plan to disable the effect of declaring elements `const` for some views such as the upcoming zip view:

```
for (const auto& elem : myZipView) {
  elem.member = value;   // WTF: C++23 plans to let this compile
}
```

Maybe we can still stop this. However, beware that making collections or their elements `const` may not work when using C++ standard views.

`const_iterator` and `cbegin()` Are Not Provided or Are Broken

Unfortunately, the usual approach of making all elements of a container `const` does not work for views:

- In general, views do ***not*** provide a `const_iterator` to do the following:

  ```
  for (decltype(rg)::const_iterator pos = rg.begin();          // not for views
       pos != range.end();
       ++pos) {
    elemfunc(*pos);
  }
  ```

  Therefore, we also cannot simply convert the range as a whole:

  ```
  std::ranges::subrange<decltype(rg)::const_iterator> crg{rg}; // not for views
  ```

- In general, views do ***not*** have the member functions `cbegin()` and `cend()` to do the following:

  ```
  callTraditionalAlgo(rg.cbegin(), rg.cend());                  // not for views
  ```

You might propose that we use the free-standing helper functions `std::cbegin()` and `std::cend()`, which have been provided since C++11 instead. They were introduced to ensure that elements are `const` when you iterate over them. Here, however, the situation is even worse, because **`std::cbegin()` and `std::cend()` are broken for views**. Their specification did not take into account types that have shallow constness (internally, they call the `const` member function `begin()` and do ***not*** add constness to the value type). Therefore, for views that do not propagate constness, these functions are simply broken:[8]

```
for (auto pos = std::cbegin(range); pos != std::cend(range); ++pos) {
  elemfunc(*pos);  // does not provide constness for values of views
}
```

Note also that when using these helper functions, there are some issues with ADL. For that reason, C++20 introduces corresponding helper functions of the ranges library in the namespace `std::ranges`. Unfor-

[8] `std::cbegin()` and `std::cend()` should simply be fixed. However, so far, the C++ standards committee has rejected each proposal to fix them.

tunately, **std::ranges::cbegin() and std::ranges::cend() are also broken for views in C++20** (this will be fixed in C++23, though):

```
for (auto pos = std::ranges::cbegin(rg); pos != std::ranges::cend(rg); ++pos) {
  elemfunc(*pos);   // OOPS: Does not provide constness for the values in C++20
}
```

Thus, in C++20, we have yet another severe problem when typing to make all elements of a view const:

> **BE CAREFUL when using cbegin(), cend(), cdata() in generic code because these functions are not available or are broken for some views.**

I do not know what is worse: that we have this problem or that the C++ standards committee and its subgroup for the ranges library were not willing to fix this problem for C++20. For example, we could provide const_iterator and cbegin() members in std::ranges::view_interface<>, which serves as a base class for all views. The funny thing is that only one view provides const_iterator support so far: std::string_view. Ironically, that is more or less the only view where we do not need it, because in a string view, the characters are always const.

However, there is some hope: http://wg21.link/p2278 provides an approach to fix this broken constness for C++23 (unfortunately, ***not*** for std::cbegin() and std::cend(), sigh).[9] With that approach, inside a generic function, you can make the elements const as follows:

```
void print(auto&& rgPassed)
{
  auto rg = std::views::as_const(rgPassed);   // ensure all elements are const
  ...   // use rg instead of rgPassed now
}
```

As another approach, you can do the following:[10]

```
void print(R&& rg)
{
  if constexpr (std::ranges::const_range<R>) {
    // the passed range and its elements are constant
    ...
  }
  else {
    // call this function again after making view and elements const:
    print(std::views::as_const(std::forward<R>(rg)));
  }
}
```

Unfortunately, C++23 will still not provide a generic way to declare a reference parameter for both containers and views that guarantees that the elements are const inside. You cannot declare a function print() that guarantees in its signature not to modify the elements.

[9] Thanks a lot to Barry Revzin who did the tremendous work to propose this fix.

[10] Thanks to Tomasz Kaminski for pointing this out.

6.6 Summary of All Container Idioms Broken By Views

As documented in this chapter, several idioms that we we can count on when using containers are broken by views. Here is a quick summary you should always take into account when using generic code for both containers and views:

- You might not be able to iterate over the elements of a standard view when it is `const`.

  As a consequence, generic code for all kinds of ranges (containers and views) has to declare the parameters as universal/forwarding references.

  However, do not use universal/forwarding references when you have concurrent iterations. In that case, `const` is your friend.
- Standard views on lvalue ranges do not propagate constness.

  This means that declaring such a view as `const` does not declare the elements to be `const`.
- Concurrent iterations over standard views might cause data races (a runtime error due to undefined behavior) even if they only read.
- Reading iterations might affect later functional behavior or even invalidate later iterations.

  Use standard views ad hoc.
- Copying a view might create a view with different states and behavior than the source view.

  Avoid copying standard views.
- `cbegin()` and `cend()` might not make elements const.

  This will partially be fixed with C++23 by providing `cbegin()` and `cend()` member functions and fixing `std::ranges::cbegin()` and `std::ranges::cend()`. Unfortunately, `std::cbegin()` and `std::cend()` will still be broken.
- Type `const_iterator` is usually not available.
- For C++23, the following breakage is planned: for some standard views, declaring the ***elements*** as `const` might have no effect. You might be able to modify the members of a `const` element of a view.

This means that a standard view is not always a pure subset that restricts or deals with the elements of a range; it might provide options and operations that are not allowed for when using the range as a whole instead.

As a consequence, use views ad hoc and with care and forget about any temporary constness. Honestly, you might consider avoiding standard views and using something with a safer design instead.

6.7 Afternotes

After adopting the **S**tandard **T**emplate **L**library for the first C++ standard, which introduced a pair of iterators as abstraction for containers/collections to deal with them in algorithms, there was always an ongoing discussion about how to deal with a single range object instead of passing begin iterators and end iterators. The Boost.Range library and the Adobe Source Libraries (ASL) were two early approaches that came up with concrete libraries.

The first proposal for ranges becoming a C++ standard was in 2005 by Thorsten Ottosen in `http://wg21.link/n1871`. Eric Niebler took on the task of driving this forward over years (supported by many other people), which resulted in another proposal in 2014 by Eric Niebler, Sean Parent, and Andrew Sutton in `http://wg21.link/n4128` (this document contains the rationale for many key design decisions). As a consequence, in October 2015, a ***Ranges T****echnical* ***S****pecification* was opened starting with `http://wg21.link/n4560`.

The ranges library was finally adopted by merging the *Ranges TS* into the C++ standard as proposed by Eric Niebler, Casey Carter, and Christopher Di Bella in `http://wg21.link/p0896r4`.

After adoption, several proposals, papers, and even defects against C++20 changed significant aspects, especially for views. See, for example, `http://wg21.link/p2210r2` (fixing split views), `http://wg21.link/p2325r3` (fixing the definition of views), `http://wg21.link/p2415r2` (introducing owning views for rvalue ranges), and `http://wg21.link/p2432r1` (fixing istream views) proposed by Barry Revzin, Tim Song, and Nicolai Josuttis.

In addition, note that at least some of the several `const` issues with views will probably be fixed in C++23 with `http://wg21.link/p2278r4`. Hopefully, vendors will provide that fix before C++23 because otherwise, C++20 code might not be compatible with C++23 here.

This page is intentionally left blank

Chapter 7
Utilities for Ranges and Views

After the introduction to ranges and views in the previous chapter, this chapter gives a first overview of important details and utilities (functions, types, and objects) that were introduced with C++20 to deal with ranges and views. In particular, this chapter covers:

- The most important general purpose utilities for creating views
- A new way to categorize iterators and the consequence of this
- New iterator and sentinel types
- New functions and type functions for dealing with ranges and iterators

At the end, this chapter also lists all algorithms that C++20 now provides with details about whether the algorithms can be used for for ranges as a whole and a few key aspects that apply then.

Note that there are other places in this book where features of ranges are described in detail:

- The general chapter about concepts presents all range concepts.
- The next chapter presents all the details of the standard view types.

7.1 Key Utilities for Using Ranges as Views

As introduced with views and the way they are used, C++ provides several ***range adaptors*** and ***range factories*** so that you can easily create views with the best performance.

Several of these adaptors apply to specific view types. However, some of them might create different things depending on the characteristics of the passed ranges. For example, adaptors might just yield the passed ranges if they already have the characteristics of the result.

There are a few key range adaptors and factories that make it easy to create views or convert ranges into views with specific characteristics (independently of the content):

- `std::views::all()` is the primary range adaptor for converting a passed range into a view.
- `std::views::counted()` is the primary range factory for converting a passed begin and count/size into a view.

- `std::views::common()` is a range adaptor that converts a range with different types for the (begin) iterator and the sentinel (end iterator) into a view with harmonized types for begin and end.

These adaptors are described in this section.

7.1.1 std::views::all()

The range adaptor `std::views::all()` is ***the*** adaptor for converting any range that is not a view yet into a view. That way, you guarantee that you have a cheap handle for dealing with the elements of a range.

`all(`*`rg`*`)` yields the following:[1]

- A copy of *`rg`* if it is already a view
- Otherwise, a `std::ranges::ref_view` of *`rg`* if it is an lvalue (range object with a name)
- Otherwise, a `std::ranges::owning_view` of *`rg`* if it is an rvalue (unnamed temporary range object or range object marked with `std::move()`)

For example:

```
std::vector<int> getColl();                         // function returning a tmp. container
std::vector coll{1, 2, 3};                          // a container
std::ranges::iota_view aView{1};                    // a view

auto v1 = std::views::all(aView);                   // decltype(coll)
auto v2 = std::views::all(coll);                    // ref_view<decltype(coll)>
auto v3 = std::views::all(std::views::all(coll));   // ref_view<decltype(coll)>
auto v4 = std::views::all(getColl());               // owning_view<decltype(coll)>
auto v5 = std::views::all(std::move(coll));         // owning_view<decltype(coll)>
```

The `all()` adaptor is usually used to pass a range as a lightweight object. Converting a range into a view can be useful for two reasons:

- Performance:

  When ranges are copied (e.g., because the parameter takes the argument by value), it is much cheaper to use a view instead. The reason for this is that moving or (if supported) copying a view is guaranteed to be cheap. Therefore, passing the range as a view makes the call cheap.

  For example:

  ```
  void foo(std::ranges::input_range auto coll)   // NOTE: takes range by value
  {
    for (const auto& elem : coll) {
      ...
    }
  }
  ```

[1] C++20 originally stated that `all()` might yield a subrange but not an owning view. However, that was fixed after C++20 came out (see `http://wg21.link/p2415`).

```
    std::vector<std::string> coll{...};

    foo(coll);                                  // OOPS: copies coll
    foo(std::views::all(coll));                 // OK: passes coll by reference
```

Using `all()` here is a bit like using reference wrappers (created with `std::ref()` or `std::cref()`). However, `all()` has the benefit that what is passed still supports the usual interface of ranges.

Passing a container to a coroutine (a coroutine usually has to take parameters by value) may be another application of this technique.

- Supporting requirements for a view:

  Another reason to use `all()` is to meet constraints that require a view. One reason for the requirement might be to ensure that passing the argument is not expensive (which leads to the benefit above).

  For example:

```
    void foo(std::ranges::view auto coll)       // NOTE: takes view by value
    {
      for (const auto& elem : coll) {
        ...
      }
    }

    std::vector<std::string> coll{...};

    foo(coll);                                  // ERROR
    foo(std::views::all(coll));                 // OK (passes coll by reference)
```

Using `all()` might happen implicitly. ***The*** example of implicit conversions with `all()` is the initialization of view types with containers.

Type `std::views::all_t<>`

C++20 also defines type `std::views::`**`all_t<>`** as the type that `all()` yields. It follows the rules of *perfect forwarding*, meaning that both a value type and an rvalue reference type can be used to specify the type of an rvalue:

```
  std::vector<int> v{0, 8, 15, 47, 11, -1, 13};
  ...
  std::views::all_t<decltype(v)>   a1{v};             // ERROR
  std::views::all_t<decltype(v)&>  a2{v};             // ref_view<vector<int>>
  std::views::all_t<decltype(v)&&> a3{v};             // ERROR
  std::views::all_t<decltype(v)>   a4{std::move(v)};  // owning_view<vector<int>>
  std::views::all_t<decltype(v)&>  a5{std::move(v)};  // ERROR
  std::views::all_t<decltype(v)&&> a6{std::move(v)};  // owning_view<vector<int>>
```

Views that take a range usually use type `std::views::all_t<>` to ensure that the passed range is really a view. For that reason, if you pass a range that is not a view yet, a view is implicitly created. For example, calling:

```
std::views::take(coll, 3)
```

has the same effect as calling:

```
std::ranges::take_view{std::ranges::ref_view{coll}, 3};
```

The way this works is as follows (using a take view as an example):

- The view type requires that a view is passed:

  ```
  namespace std::ranges {
    template<view V>
    class take_view : public view_interface<take_view<V>> {
     public:
      constexpr take_view(V base, range_difference_t<V> count);
      ...
    };
  }
  ```

- A deduction guide requires that the element type of the view is of type `std::views::all_t`:

  ```
  namespace std::ranges {
    // deduction guide to force the conversion to a view:
    template<range R>
    take_view(R&&, range_difference_t<R>) -> take_view<views::all_t<R>>;
  }
  ```

- Now when a container is passed, the range type of the view has to have type `std::views::all_t<>`. Therefore, the container is implicitly converted to a `ref_view` (if it is an lvalue) or an `owning_view` (if it is an rvalue). The effect is as if we had called the following:

  ```
  std::ranges::ref_view rv{coll};      // convert to a ref_view<>
  std::ranges::take_view tv(rv, 3);    // and use view and count to initialize the take_view<>
  ```

The concept `viewable_range` can be used to check whether a type can be used for `all_t<>` (and therefore corresponding objects can be passed to `all()`):[2]

```
std::ranges::viewable_range<std::vector<int>>               // true
std::ranges::viewable_range<std::vector<int>&>              // true
std::ranges::viewable_range<std::vector<int>&&>             // true
std::ranges::viewable_range<std::ranges::iota_view<int>>    // true
std::ranges::viewable_range<std::queue<int>>                // false
```

7.1.2 `std::views::counted()`

The range factory `std::views::`**`counted()`** provides the most flexible way to create a view from a begin iterator and a count. By calling

```
std::views::counted(beg, sz)
```

it creates a view to the first `sz` elements of the range starting with `beg`.

[2] For lvalues of move-only view types, there is an issue that `all()` is ill-formed although `viewable_range` is satisfied.

It is up to the programmer to ensure that the begin and count are valid when the view is used. Otherwise, the program has undefined behavior.

The count may be 0, which means that the range is empty.

The count is stored in the view and is therefore stable. Even if new elements are inserted in the range that the view refers to, the count does not change. For example:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};

auto c = std::views::counted(lst.begin(), 5);
print(c);     // 1 2 3 4 5

lst.insert(++lst.begin(), 0);          // insert new second element in lst
print(c);     // 1 0 2 3 4
```

Note that if you want to have a view to the first *num* elements of a range, the take view provides a more convenient and safer way to get it:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};
auto v1 = std::views::take(lst, 3);     // view to first three elements (if they exist)
```

The take view ***does*** check whether there are enough elements and yields fewer elements if there are not.

By also using the drop view, you could even skip a specific number of leading elements:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};
auto v2 = std::views::drop(lst, 2) | std::views::take(3); // 3rd to 5th element (if ex-
ist)
```

Type of `counted()`

`counted(`*`beg`*`, `*`sz`*`)` yields different types depending on the characteristics of the range it is called for:

- It yields a `std::span` if the passed begin iterator is a `contiguous_iterator` (refers to elements that are stored in contiguous memory). This applies to raw pointers, raw arrays, and iterators of `std::vector<>` or `std::array<>`.
- Otherwise, it yields a `std::ranges::subrange` if the passed begin iterator is a `random_access_iterator` (supports jumping back and forth between elements). This applies to iterators of `std::deque<>`.
- Otherwise, it yields a `std::ranges::subrange` with a `std::counted_iterator` as the begin and a dummy sentinel of type `std::default_sentinel_t` as the end. This means that the iterator in the subrange counts while iterating. This applies to iterators of lists, associative containers, and unordered containers (hash tables).

For example:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vec5 = std::ranges::find(vec, 5);
auto v1 = std::views::counted(vec5, 3);   // view to element 5 and two more elems in vec
    // v1 is std::span<int>
```

```
std::deque<int> deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto deq5 = std::ranges::find(deq, 5);
auto v2 = std::views::counted(deq5, 3);   // view to element 5 and two more elems in deq
   // v2 is std::ranges::subrange<std::deque<int>::iterator>

std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lst5 = std::ranges::find(lst, 5);
auto v3 = std::views::counted(lst5, 3);   // view to element 5 and two more elems in lst
   // v3 is std::ranges::subrange<std::counted_iterator<std::list<int>::iterator>,
   //                              std::default_sentinel_t>
```

Note that this code is at risk because it creates undefined behavior if there is no 5 in the collection with two elements behind it. Therefore, you should check this if you are not sure that this is the case.

7.1.3 `std::views::common()`

The range adaptor `std::views::`**`common()`** yields a view with harmonized types for the begin iterator and the sentinel (end iterator) for the passed range. It acts like the range adaptor `std::views::all()`, with the additional behavior of creating a `std::ranges::common_view` from the passed argument if its iterators have different types.

For example, assume we want to call a traditional algorithm `algo()` that requires the same type for both the begin iterator and the end iterator. We can then call it for iterators of potentially different types as follows:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end());   // assume algo() requires iterators with the same type
}
```

`common(`*`rg`*`)` yields

- A copy of *rg* if it is already a view with the same begin and end iterator type
- Otherwise, a `std::ranges::ref_view` of *rg* if it is a range object with the same begin and end iterator type
- Otherwise, a `std::ranges::common_view` of *rg*

For example:

```
std::list<int> lst;
std::ranges::iota_view iv{1, 10};
...
auto v1 = std::views::common(lst);    // std::ranges::ref_view<decltype(lst)>
auto v2 = std::views::common(iv);     // decltype(iv)
auto v3 = std::views::common(std::views::all(vec));
                                      // std::ranges::ref_view<decltype(lst)>
```

```
std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vt = std::views::take(lst, 5);  // begin() and end() have different types
auto v4 = std::views::common(vt);    // std::ranges::common_view<decltype(vt)>
```

Note that both the constructors of a `std::ranges::common_view` and the helper type `std::common:iterator` require that passed iterators have different types. Therefore, you should use this adaptor if you do not know whether iterator types differ.

7.2 New Iterator Categories

Iterators have different abilities. These abilities are important because some algorithms require special iterator abilities. For example, sorting algorithms require iterators that can perform random access because otherwise, the performance would be poor. For this reason, iterators have different categories, which were extended in C++20 by a new category: *contiguous iterator*. The abilities of these categories are listed in table *Iterator categories*.

Iterator Category	Ability	Providers
Output	Writes forward	ostream iterator, inserter
Input	Reads forward once	istream iterator
Forward	Reads forward	`std::forward_list<>`, unordered containers
Bidirectional	Reads forward and backward	`list<>`, `set<>`, `multiset<>`, `map<>`, `multimap<>`
Random-access	Reads with random access	`deque<>`
Contiguous	Reads elements stored in contiguous memory	`array<>`, `vector<>`, `string`, C-style array

Table 7.1. Iterator categories

Note that significant details of the categories changed with C++20. The most important changes are:

- The new iterator category ***contiguous***. For this category, a new iterator tag type is defined: `std::`**`contiguous_iterator_tag`**.
- Iterators that yield temporary objects (prvalues) can now have a stronger category than **input iterators**. The requirement to yield references no longer applies to forward iterators.
- Input iterators are no longer guaranteed to be copyable. You should only move them around.
- Postfix increment operators of input iterators are no longer required to yield anything. For an input iterator `pos`, you should no longer use `pos++`. Use `++pos` instead.

These changes are not backward compatible:

- A check whether an iterator ***is*** a random-access iterator is no longer true for iterators of containers like `std::vector` or arrays. Instead, you have to check whether the iterator ***supports*** random access.
- The assumption that the values of forward, bidirectional, or random-access iterators are references no longer always applies.

For that reason, C++20 introduces a new ***optional*** iterator attribute, `iterator_concept`, and defines that this attribute *may* be set to signal a C++20 category that differs from the traditional category. For example:

```
std::vector vec{1, 2, 3, 4};
auto pos1 = vec.begin();
decltype(pos1)::iterator_category    // type std::random_access_iterator_tag
decltype(pos1)::iterator_concept     // type std::contiguous_iterator_tag

auto v = std::views::iota(1);
auto pos2 = v.begin();
decltype(pos2)::iterator_category    // type std::input_iterator_tag
decltype(pos2)::iterator_concept     // type std::random_access_iterator_tag
```

Note that `std::iterator_traits` do not provide a member `iterator_concept` and that for iterators, a member `iterator_category` may not always be defined.

The iterator concepts and the range concepts take the new C++20 categorization into account. For code that has to deal with iterator categories, this has the following consequences since C++20:[3]

- Use iterator concepts and range concepts instead of `std::iterator_traits<I>::iterator_category` to check the category.
- Think about providing `iterator_category` and/or `iterator_concept` according to the hints in `http://wg21.link/p2259` if you implement your own iterator types.

Note also that valid C++20 input iterators may not be C++17 iterators at all (e.g., by not providing copying). For these iterators, the traditional iterator traits do not work. For this reason, since C++20:[4]

- Use `std::iter_value_t`
  instead of `iterator_traits<>::value_type`.
- Use `std::iter_reference_t`
  instead of `iterator_traits<>::reference`.
- Use `std::iter_difference_t`
  instead of `iterator_traits<>::difference_type`.

7.3 New Iterator and Sentinel Types

For (better) support for ranges and views, C++20 introduces a couple of new iterator and sentinel types:

- `std::`**`counted_iterator`** for an iterator that itself has a *count* to specify the end of a range
- `std::`**`common_iterator`** for a common iterator type that can be used for two iterators that have different types
- `std::`**`default_sentinel_t`** for an end iterator that forces an iterator to check for its end
- `std::`**`unreachable_sentinel_t`** for an end iterator that can never be reached, meaning that the range is endless
- `std::`**`move_sentinel`** for an end iterator that maps copies to moves

[3] Thanks to Barry Revzin for pointing this out in `http://stackoverflow.com/questions/67606563`.

[4] Thanks to Hui Xie for pointing this out.

7.3.1 `std::counted_iterator`

A *counted iterator* is an iterator that has a counter for the maximum number of elements to iterate over.

There are two ways to use such an iterator:

- Iterating while checking how many elements are left:

```
for (std::counted_iterator pos{coll.begin(), 5}; pos.count() > 0; ++pos) {
  std::cout << *pos << '\n';
}
std::cout << '\n';
```

- Iterating while comparing with a default sentinel:

```
for (std::counted_iterator pos{coll.begin(), 5};
     pos != std::default_sentinel; ++pos) {
  std::cout << *pos << '\n';
}
```

This feature is used when the view adaptor `std::ranges::counted()` yields a subrange for iterators that are not random-access iterators.

Table *Operations of class* `std::counted_iterator<>` lists the API of a counted iterator.

Operation	**Effect**
countedItor *pos*{}	Creates a counted iterator that refers to no element (*count* is 0)
countedItor *pos*{*pos2*, *num*}	Creates a counted iterator of *num* elements starting with *pos2*
countedItor *pos*{*pos2*}	Creates a counted iterator as a (type converted) copy of a counted iterator *pos2*
pos`.count()`	Yields how many elements are left (0 means we are at the end)
pos`.base()`	Yields (a copy of) the underlying iterator
...	All standard iterator operations of the underlying iterator type
pos == `std::default_sentinel`	Yields whether the iterator is at the end
pos != `std::default_sentinel`	Yields whether the iterator is not at the end

Table 7.2. Operations of class `std::counted_iterator<>`

It is up to the programmer to ensure that:

- The initial *count* is not higher than the iterator initially passed
- The counted iterator does not increment more than *count* times
- The counted iterator does not access elements beyond the *count*-th element (as usual, the position behind the last element is valid)

7.3.2 `std::common_iterator`

Type `std::common_iterator<>` is used to harmonize the type of two iterators. It wraps the two iterators so that from the outside, both iterators have the same type. Internally, a value of one of the two types is stored (for this, the iterator typically uses a `std::variant<>`).

For example, traditional algorithms that take a begin iterator and an end iterator require that these iterators have the same type. If you have iterators of different types, you can use this type function be able for calling these algorithm:

```
algo(beg, end);        // if this is an error due to different types

algo(std::common_iterator<decltype(beg), decltype(end)>{beg},    // OK
     std::common_iterator<decltype(beg), decltype(end)>{end});
```

Note that it is a compile-time error if the types passed to `common_iterator<>` are the same. Thus, in generic code, you might have to call:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
  if constexpr(std::same_as<BegT, EndT>) {
    algo(beg, end);
  }
  else {
    algo(std::common_iterator<decltype(beg), decltype(end)>{beg},
         std::common_iterator<decltype(beg), decltype(end)>{end});
  }
}
```

A more convenient way to have the same effect is to use the `common()` range adaptor:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end());
}
```

7.3.3 `std::default_sentinel`

A *default sentinel* is an iterator that provides no operations at all. C++20 provides a corresponding object `std::default_sentinel`, which has type `std::default_sentinel_t`. It is provided in `<iterator>`. The type has no members:

```
namespace std {
  class default_sentinel_t {
  };
  inline constexpr default_sentinel_t default_sentinel{};
}
```

The type and the value are provided to serve as a dummy sentinel (end iterator), when an iterator knows its end without looking at the end iterator. For those iterators, a comparison with `std::default_sentinel` is defined but the default sentinel is never used. Instead, the iterator checks internally whether it is at the end (or how far it is from the end).

For example, counted iterators define an operator == for themselves with a default sentinel to perform checking whether they are at the end:

```
namespace std {
  template<std::input_or_output_iterator I>
  class counted_iterator {
    ...
    friend constexpr bool operator==(const counted_iterator& p,
                                     std::default_sentinel_t) {
      ...   // returns whether p is at the end
    }
  };
}
```

This allows code like this:

```
// iterate over the first five elements:
for (std::counted_iterator pos{coll.begin(), 5};
     pos != std::default_sentinel;
     ++pos) {
  std::cout << *pos << '\n';
}
```

The standard defines the following operations with default sentinels:

- For `std::counted_iterator`:
  - Comparisons with operator == and `!=`
  - Computing the distance with operator `-`
- `std::views::counted()` may create a subrange of a counted iterator and a default sentinel
- For `std::istream_iterator`:
  - Default sentinels can be used as an initial value, which has the same effect as the default constructor
  - Comparisons with operator == and `!=`
- For `std::istreambuf_iterator`:
  - Default sentinels can be used as an initial value, which has the same effect as the default constructor
  - Comparisons with operator == and `!=`
- For `std::ranges::basic_istream_view<>`:
  - Istream views yield `std::default_sentinel` as `end()`
  - Istream view iterators can compare with operator == and `!=`
- For `std::ranges::take_view<>`:
  - Take views may yield `std::default_sentinel` as `end()`
- For `std::ranges::split_view<>`:
  - Split views may yield `std::default_sentinel` as `end()`
  - Split view iterators can compare with operator == and `!=`

7.3.4 `std::unreachable_sentinel`

The value `std::unreachable_sentinel` of type `std::unreachable_sentinel_t` was introduced in C++20 to specify a sentinel (end iterator of a range) that is not reachable. It effectively says "*Do not compare with me at all*." This type and value can be used to specify unlimited ranges.

When it is used, it can optimize generated code because a compiler can detect that comparing it with another iterator never yields `true`, meaning that it might skip any check against the end.

For example, ***if we know*** that a value `42` exists in a collection, we can search for it as follows:

```
auto pos42  = std::ranges::find(coll.begin(), std::unreachable_sentinel,
                                42);
```

Normally, the algorithm would compare against both, `42` and `coll.end()` (or whatever is passed as the end/sentinel). Because `unreachable_sentinel` is used, for which any comparison with an iterator always yields `false`, compilers can optimize the code to compare only against `42`. Of course, programmers have to ensure that a `42` exists.

The iota view provides an example of using `unreachable_sentinel` for endless ranges.

7.3.5 `std::move_sentinel`

Since C++11, the C++ standard library has an iterator type `std::move_iterator`, which can be used to map the behavior of an iterator from copying to moving values. C++20 introduces the corresponding sentinel type.

This type can only be used to compare a move sentinel with a move iterator using operators `==` and `!=` and to compute the difference between a move iterator and a move sentinel (if supported).

If you have an iterator and a sentinel that form a valid range (concept `std::sentinel_for` is satisfied), you can convert them into a move iterator and a move sentinel to get a valid range for which `sentinel_for` is still met.

You can use the move sentinel as follows:

```
std::list<std::string> coll{"tic", "tac", "toe"};
std::vector<std::string> v1;
std::vector<std::string> v2;

// copy strings into v1:
for (auto pos{coll.begin()}; pos != coll.end(); ++pos) {
  v1.push_back(*pos);
}

// move strings into v2:
for (std::move_iterator pos{coll.begin()};
       pos !=  std::move_sentinel(coll.end()); ++pos) {
  v2.push_back(*pos);
}
```

Note that this has the effect that the iterator and the sentinel have different types. Therefore, you cannot initialize the vector directly because it requires the same type for begin and end. In that case, you also have to use a move iterator for the end:

```
std::vector<std::string> v3{std::move_iterator{coll.begin()},
                            std::move_sentinel{coll.end()}};    // ERROR

std::vector<std::string> v4{std::move_iterator{coll.begin()},
                            std::move_iterator{coll.end()}};    // OK
```

7.4 New Functions for Dealing with Ranges

The ranges library provides several generic helper functions in the namespaces `std::ranges` and `std`.

Several of them already existed before C++20 with the same name or a slightly different name in the namespace `std` (and are still provided for backward compatibility). However, the range utilities usually provide better support for the specified functionality. They might fix flaws that the old versions have or use concepts to constrain their use.

These "functions" are required to be *function objects* or *functors* or even ***Customization Point Objects*** (CPOs are function objects that are required to be semiregular and guarantee that all instances are equal). This means that they do not support argument-dependent lookup (ADL).[5]

For this reason, **you should prefer `std::ranges` utilities over `std` utilities.**

7.4.1 Functions for Dealing with the Elements of Ranges (and Arrays)

Table *Generic functions for dealing with the elements of ranges* lists the new free-standing functions for dealing with ranges and their elements. They also work for raw arrays.

Almost all corresponding utility functions were also available before C++20 directly in the namespace `std`. The only exceptions are:

- `ssize()`, which is introduced with C++20 as `std::ssize()`
- `cdata()`, which did not and does not exist in the namespace `std` at all

You might wonder why there are new functions in the namespace `std::ranges` instead of fixing the existing functions in the namespace `std`. Well, the argument is that if we fix the existing functions, existing code might be broken and backward compatibility is an important goal of C++.

The next question is then which function to use when. The guideline here is pretty simple: **prefer the functions/utilities in the namespace `std::ranges` over those in the namespace `std`**.

The reason is not only that the functions/utilities in the namespace `std::ranges` use concepts, which helps to find problems and bugs at compile time; another reason to prefer the new functions in the namespace `std::ranges` is that the functions in the namespace `std` sometimes have flaws, which the new implementation in `std::ranges` fixes:

- One problem is *argument-dependent lookup (ADL)*.
- Another problem is `const` correctness.

[5] You can argue that it is not correct to call function objects just functions. However, because they are used like functions, I still tend to do so. The rest are implementation details.

Function	Meaning
`std::ranges::`**`empty`**`(`*rg*`)`	Yields whether the range is empty
`std::ranges::`**`size`**`(`*rg*`)`	Yields the size of the range
`std::ranges::`**`ssize`**`(`*rg*`)`	Yields the size of the range as the value of a signed type
`std::ranges::`**`begin`**`(`*rg*`)`	Yields an iterator to the first element of the range
`std::ranges::`**`end`**`(`*rg*`)`	Yields a sentinel (an iterator to the end) of the range
`std::ranges::`**`cbegin`**`(`*rg*`)`	Yields a constant iterator to the first element of the range
`std::ranges::`**`cend`**`(`*rg*`)`	Yields a constant sentinel (a constant iterator to the end) of the range
`std::ranges::`**`rbegin`**`(`*rg*`)`	Yields a reverse iterator to the first element of the range
`std::ranges::`**`rend`**`(`*rg*`)`	Yields a reverse sentinel (an iterator to the end) of the range
`std::ranges::`**`crbegin`**`(`*rg*`)`	Yields a reverse constant iterator to the first element of the range
`std::ranges::`**`crend`**`(`*rg*`)`	Yields a reverse constant sentinel (a constant iterator to the end) of the range
`std::ranges::`**`data`**`(`*rg*`)`	Yields the raw data of the range
`std::ranges::`**`cdata`**`(`*rg*`)`	Yields the raw data of the range with `const` elements

Table 7.3. Generic functions for dealing with the elements of ranges

How `std::ranges::begin()` Solves ADL Problems

Let us look at why using `std::ranges::begin()` is better than using `std::begin()` to deal with ranges in generic code. The problem is that argument-dependent lookup does not always work for `std::begin()` if you do not use it in a tricky way.[6]

Assume you want to write generic code that calls the `begin()` function defined for a range object `obj` of an arbitrary type. The issue we have with `std::begin()` is as follows:

- To support range types like containers that have a `begin()` member function, we can always call the member function `begin()`:

  ```
  obj.begin();          // only works if a member function begin() is provided
  ```

  However, that does not work for types that have only a free-standing `begin()` (such as raw arrays).
- Unfortunately, if we use the free-standing `begin()`, we have a problem:
  - The standard free-standing `std::begin()` for raw arrays needs the full qualification with `std::`.
  - Other non-standard ranges cannot deal with the full qualification `std::`. For example:

    ```
    class MyColl {
      ...
    };
    ... begin(MyColl);   // declare free-standing begin() for MyColl

    MyColl obj;
    std::begin(obj);   // std::begin() does not find ::begin(MyType)
    ```

[6] Thanks to Tim Song and Barry Revzin for pointing this out.

- The necessary workaround is to put an additional `using` declaration in front of the `begin()` call and ***not*** qualify the call itself:

  ```
  using std::begin;
  begin(obj);          // OK, works in all these cases
  ```

The new `std::ranges::begin()` does not have this problem:

```
std::ranges::begin(obj);    // OK, works in all these cases
```

The trick is that `begin` is not a function that uses argument-dependent lookup, but rather a function object that implements all possible lookups. See *`lib/begin.cpp`* for a complete example.

And that applies to all utilities that are defined in `std::ranges`. Therefore, code that uses `std::ranges` utilities generally supports more types and more complex use cases.

7.4.2 Functions for Dealing with Iterators

Table *Generic functions for dealing with iterators* lists all generic functions for moving iterators, looking ahead or back, and computing the distance between iterators. Note that the next subsection describes functions for swapping and moving elements that iterators refer to.

Function	**Meaning**
`std::ranges::`**`distance`**`(`*from*`,` *to*`)`	Yields the distance (number of elements) between *from* and *to*
`std::ranges::`**`distance`**`(`*rg*`)`	Yields the number of elements in *rg* (size even for ranges that have no `size()`)
`std::ranges::`**`next`**`(`*pos*`)`	Yields the position of the next element behind *pos*
`std::ranges::`**`next`**`(`*pos*`,` *n*`)`	Yields the position of the *n*-th next element behind *pos*
`std::ranges::`**`next`**`(`*pos*`,` *to*`)`	Yields the position *to* behind *pos*
`std::ranges::`**`next`**`(`*pos*`,` *n*, *maxpos*`)`	Yields the position of the *n*-th element after *pos* but not behind *maxpos*
`std::ranges::`**`prev`**`(`*pos*`)`	Yields the position of the element before *pos*
`std::ranges::`**`prev`**`(`*pos*`,` *n*`)`	Yields the position of the *n*-th element before *pos*
`std::ranges::`**`prev`**`(`*pos*`,` *n*, *minpos*`)`	Yields the position of the *n*-th element before *pos* but not before *minpos*
`std::ranges::`**`advance`**`(`*pos*`,` *n*`)`	Advances *pos* forward/backward *n* elements
`std::ranges::`**`advance`**`(`*pos*`,` *to*`)`	Advances *pos* forward to *to*
`std::ranges::`**`advance`**`(`*pos*`,` *n*, *maxpos*`)`	Advances *pos* forward/backward *n* elements but not further than *maxpos*

Table 7.4. Generic functions for dealing with iterators

Again, you should prefer these utilities over the corresponding traditional utilities in the namespace `std`. For example, in contrast to `std::ranges::next()`, the old utility `std::next()` requires to provide `std::iterator_traits<It>::difference_type` for a passed iterator `It`. However, some internal iterators of view types do not support iterator traits, so that code might not compile, if you use `std::next()`.

7.4.3 Functions for Swapping and Moving Elements/Values

The ranges library also provides functions for swapping and moving values. These functions are listed in table *Generic functions for swapping and moving elements/values*. They cannot only be used for ranges.

Function	**Meaning**
`std::ranges::`**`swap`**`(`*val1*`,` *val2*`)`	Swaps the values *val1* and *val2* (using move semantics)
`std::ranges::`**`iter_swap`**`(`*pos1*`,` *pos2*`)`	Swaps the values that iterators *pos1* and *pos2* refer to (using move semantics)
`std::ranges::`**`iter_move`**`(`*pos*`)`	Yields the value that iterator *pos* refers to for a move

Table 7.5. Generic functions for swapping and moving elements/values

The function `std::ranges::swap()` is defined in `<concepts>` (`std::swap()` is defined in `<utility>`) because the standard concepts `std::swappable` and `std::swappable_with` use it. The functions `std::ranges::iter_swap()` and `std::ranges::iter_move()` are defined in `<iterator>`.

`std::ranges::swap()` fixes the problem that calling `std::swap()` in generic code might not find the best swap function provided for certain types (similar to the problems that `begin()` and `cbegin()` have). However, because there is a generic fallback, this fix does not fix code that does not compile. Instead, it may just create better performance.

Consider the following example:

lib/swap.cpp

```cpp
#include <iostream>
#include <utility>    // for std::swap()
#include <concepts>   // for std::ranges::swap()

struct Foo {
  Foo() = default;

  Foo(const Foo&) {
    std::cout << "  COPY constructor\n";
  }
  Foo& operator=(const Foo&) {
    std::cout << "  COPY assignment\n";
    return *this;
  }

  void swap(Foo&) {
    std::cout << "  efficient swap()\n";   // swaps pointers, no data
  }
};

void swap(Foo& a, Foo& b) {
  a.swap(b);
}
```

```
int main()
{
  Foo a, b;

  std::cout << "--- std::swap()\n";
  std::swap(a, b);              // generic swap called

  std::cout << "--- swap() after using std::swap\n";
  using std::swap;
  swap(a, b);                   // efficient swap called

  std::cout << "--- std::ranges::swap()\n";
  std::ranges::swap(a, b);      // efficient swap called
}
```

The program has the following output:

```
--- std::swap()
 COPY constructor
 COPY assignment
 COPY assignment
--- swap() after using std::swap
 efficient swap()
--- std::ranges::swap()
 efficient swap()
```

Note that using `std::ranges::swap()` is counter-productive when swap functions are defined in the namespace `std` for types *not* defined in `std` (which is formally not allowed):

```
class Foo {
  ...
};

namespace std {
  void swap(Foo& a, Foo& b) {   // not found by std::ranges::swap()
    ...
  }
}
```

You should change this code. The best option is to use a “hidden friend:”

```
class Foo {
  ...
  friend void swap(Foo& a, Foo& b) {   // found by std::ranges::swap()
    ...
  }
};
```

The functions `std::ranges::iter_move()` and `std::ranges::iter_swap()` have one benefit over dereferencing the iterators and calling `std::move()` or `swap()`: they work fine with proxy iterators. Therefore, to support proxy iterators in generic code, use:

```
auto val = std::ranges::iter_move(it);
std::ranges::iter_swap(it1, it2);
```

rather than:

```
auto val = std::move(*it);
using std::swap;
swap(*it1, *it2);
```

7.4.4 Functions for Comparisons of Values

Table *Generic functions for comparison criteria* lists all range utilities that you can use as comparison criteria now. They are defined in `<functional>`.

Function	**Meaning**
`std::ranges::`**`equal_to`**`(`*val1*, *val2*`)`	Yields whether *val1* is equal to *val2*
`std::ranges::`**`not_equal_to`**`(`*val1*, *val2*`)`	Yields whether *val1* is not equal to *val2*
`std::ranges::`**`less`**`(`*val1*, *val2*`)`	Yields whether *val1* is less than *val2*
`std::ranges::`**`greater`**`(`*val1*, *val2*`)`	Yields whether *val1* is greater than *val2*
`std::ranges::`**`less_equal`**`(`*val1*, *val2*`)`	Yields whether *val1* is less than or equal to *val2*
`std::ranges::`**`greater_equal`**`(`*val1*, *val2*`)`	Yields whether *val1* is greater that or equal to *val2*

Table 7.6. Generic functions for comparison criteria

When checking for equality, the concept `std::equality_comparable_with` is used.

When checking for an order, the concept `std::totally_ordered_with` is used. Note that this does not require that `operator<=>` has to be supported. Note also that the type does not have to support a total order. It is enough if you can compare the values with <, >, <=, and >=. However, this is a semantic constraint that cannot be checked at compile time.

Note that C++20 also introduces the function object type `std::compare_three_way`, which you can use to call the new operator <=>.

7.5 New Type Functions/Utilities for Dealing with Ranges

This section lists all new auxiliary type functions and utilities provided by the ranges library as part of C++20.

As for the generic helper functions, several of these utilities already existed before C++20 with the same name or a slightly different name in the namespace `std` (and are still provided for backward compatibility). However, the range utilities usually provide better support for the specified functionality. They might fix flaws that the old versions have or use concepts to constrain their use.

7.5.1 Generic Types of Ranges

Table *Generic functions that yield the types involved when using ranges* lists all generic type definitions for ranges. They are defined as alias templates.

Type Function	Meaning
`std::ranges::`**`iterator_t`**`<`*Rg*`>`	Type of an iterator that iterates over *Rg* (what `begin()` yields)
`std::ranges::`**`sentinel_t`**`<`*Rg*`>`	Type of an end iterator for *Rg* (what `end()` yields)
`std::ranges::`**`range_value_t`**`<`*Rg*`>`	Type of the element in the range
`std::ranges::`**`range_reference_t`**`<`*Rg*`>`	Type of a reference to the element type
`std::ranges::`**`range_difference_t`**`<`*Rg*`>`	Type of the difference between two iterators
`std::ranges::`**`range_size_t`**`<`*Rg*`>`	Type of what the `size()` function returns
`std::ranges::` **`range_rvalue_reference_t`**`<`*Rg*`>`	Type of an rvalue reference to the element type
`std::ranges::`**`borrowed_iterator_t`**`<`*Rg*`>`	`std::ranges::iterator_t<`*Rg*`>` for a borrowed range, otherwise `std::ranges::dangling`
`std::ranges::`**`borrowed_subrange_t`**`<`*Rg*`>`	The subrange type of the iterator type for a borrowed range, otherwise `std::ranges::dangling`

Table 7.7. Generic functions that yield the types involved when using ranges

As mentioned in the introduction to ranges, the key benefit of these type functions is that they work generically for all types of ranges and views. This applies even to raw arrays.

Note that `std::ranges::range_value_t<`*Rg*`>` is just a shortcut for:
`std::iter_value_t<std::ranges::iterator_t<`*Rg*`>>`.
It should be preferred over *Rg*`::value_type`.

7.5.2 Generic Types of Iterators

The ranges library also introduced new type traits for iterators. Note that these traits are *not* defined in the namespace `std::ranges`, but rather in the namespace `std`.

Table *Generic functions that yield the types involved when using iterators* lists all generic type traits for iterators. They are defined as alias templates.

Type Function	Meaning
`std::`**`iter_value_t`**`<`*It*`>`	Type of the value that the iterator refers to
`std::`**`iter_reference_t`**`<`*It*`>`	Type of a reference to the value type
`std::`**`iter_rvalue_reference_t`**`<`*It*`>`	Type of an rvalue reference to the value type
`std::`**`iter_common_reference_t`**`<`*It*`>`	The common type of the reference type and a reference to the value type
`std::`**`iter_difference_t`**`<`*It*`>`	Type of the difference between two iterators

Table 7.8. Generic functions that yield the types involved when using iterators

Note that due to better support for the new iterator categories, you should prefer these utilities over traditional iterator traits (`std::iterator_traits<>`).

These functions are described in detail in the section about iterator traits.

7.5.3 New Functional Types

Table *New functional types* lists the new types that can be used as helper functions. They are defined in `<functional>`.

Type Function	Meaning
`std::`**`identity`**	A function object that returns itself
`std::`**`compare_three_way`**	A function object to call the operator `<=>`

Table 7.9. New functional types

The function object `std::identity` is usually used to pass no projection where a projection can be passed. If an algorithm supports passing a projection:

```
auto pos = std::ranges::find(coll, 25,                          // find value 25
                             [](auto x) {return x*x;});          // for squared elements
```

using `std::identity` enables programmers to pass “no projection:”

```
auto pos = std::ranges::find(coll, 25,                          // find value 25
                             std::identity{});                   // for elements as they are
```

The function object is used as a default template parameter to declare an algorithm where the projection can be skipped:

```
template<std::ranges::input_range R,
         typename T,
         typename Proj = std::identity>
constexpr std::ranges::borrowed_iterator_t<R>
find(R&& r, const T& value, Proj proj = {});
```

The function object type `std::compare_three_way` is a new function object type for specifying that the new operator `<=>` shall be called/used (just like `std::less` or `std::ranges::less` stand for calling the operator `<`). It is described in the chapter about the operator `<=>`.

7.5.4 Other New Types for Dealing with Iterators

Table *Other new types for iterators* lists all new types for dealing with iterators. They are defined in `<iterator>`.

Type Function	**Meaning**
`std::`**`incrementable_traits<`***It***`>`**	A helper type to yield the `difference_type` of two iterators
`std::`**`projected<`***It1*`,`*It2***`>`**	A type to formulate constraints for projections

Table 7.10. Other new types for iterators

Note also the new iterator and sentinel types.

7.6 Range Algorithms

With the support to pass ranges as one argument and be able to process sentinels (end iterators) with a different type, new ways of calling algorithms are now available in C++20.

However, they come with some restrictions and not all algorithms support passing ranges as a whole yet.

7.6.1 Benefits and Restrictions for Range Algorithms

For some algorithms, there are no range versions that allow programmers to pass ranges as a single argument yet:

- Range algorithms do not support parallel execution (introduced with C++17). There are no APIs that take range parameters as single objects and an execution policy.
- There are no numeric algorithms for single-object ranges yet. To call algorithms like `std::accumulate()`, you still have to pass both the begin and the end of the range. C++23 will probably provide numeric range algorithms.

If there is range support for algorithms, they use concepts to find possible errors at compile time:

- Concepts for iterators and ranges ensure that you pass valid iterators and ranges.
- Concepts for callables ensure that you pass valid helper functions.

In addition, return types may differ because:

- They support and might return iterators of different types, for which special return types are defined. These types are listed in table *New return types for range algorithms*.
- They might return borrowed iterators, signaling that the iterator is no valid iterator because a temporary range (rvalue) was passed.

Type	**Meaning**	**Members**
`std::ranges::`**`in_in_result`**	For the positions of two input ranges	`in1`, `in2`
`std::ranges::`**`in_out_result`**	For one position of an input range and one position of an output range	`in`, `out`
`std::ranges::`**`in_in_out_result`**	For the positions of two input ranges and one position of an output range	`in1`, `in2`, `out`
`std::ranges::`**`in_out_out_result`**	For one position of an input range and the position of two output ranges	`in`, `out1`, `out2`
`std::ranges::`**`in_fun_result`**	For one position of an input range and a function	`in`, `out`
`std::ranges::`**`min_max_result`**	For one maximum and one minimum position/value	`min`, `max`
`std::ranges::`**`in_found_result`**	For one position of an input range and a Boolean value	`in`, `found`

Table 7.11. New return types for range algorithms

The following program demonstrates how to use these return types:

ranges/results.cpp

```cpp
#include <iostream>
#include <string_view>
#include <vector>
#include <algorithm>

void print(std::string_view msg, auto beg, auto end)
{
   std::cout << msg;
   for(auto pos = beg; pos != end; ++pos) {
      std::cout << ' ' << *pos;
   }
   std::cout << '\n';
}

int main()
{
    std::vector inColl{1, 2, 3, 4, 5, 6, 7};
    std::vector outColl{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto result = std::ranges::transform(inColl, outColl.begin(),
                                         [] (auto val) {
                                           return val*val;
                                         });

    print("processed in:", inColl.begin(), result.in);
    print("rest of in:  ", result.in, inColl.end());
    print("written out: ", outColl.begin(), result.out);
    print("rest of out: ", result.out, outColl.end());
}
```

The program has the following output:

```
processed in: 1 2 3 4 5 6 7
rest of in:
written out:  1 4 9 16 25 36 49
rest of out:  8 9 10
```

7.6.2 Algorithm Overview

The situation of all algorithms becomes more and more complex. Some of them can be used in parallel, some not. Some of them are supported by the ranges library, some not. This section gives an overview of which algorithms are available in which form (without going into detail about what each algorithm does).

The last three columns have the following meaning:

- **Ranges** signals whether the algorithm is supported by the range library
- **`_result`** signals whether and which **`_result`** types are used (e.g., `in_out` stands for `in_out_result`)
- **Borrowed** signals whether the algorithm returns borrowed iterators

Table *Non-modifying algorithms* lists the available non-modifying standard algorithms of C++20.

Name	**Since**	**Parallel**	**Ranges**	**`_result`**	**Borrowed**
`for_each()`	C++98	yes	yes	`in_fun`	yes
`for_each_n()`	C++17	yes	yes	`in_fun`	
`count()`	C++98	yes	yes		
`count_if()`	C++98	yes	yes		
`min_element()`	C++98	yes	yes	yes	
`max_element()`	C++98	yes	yes	yes	
`minmax_element()`	C++11	yes	yes	`min_max`	yes
`min()`	C++20	no	yes (only)		
`max()`	C++20	no	yes (only)		
`minmax()`	C++20	no	yes (only)	`min_max`	
`find()`	C++98	yes	yes		yes
`find_if()`	C++98	yes	yes		yes
`find_if_not()`	C++11	yes	yes		yes
`search()`	C++98	yes	yes		yes
`search_n()`	C++98	yes	yes		yes
`find_end()`	C++98	yes	yes		yes
`find_first_of()`	C++98	yes	yes		yes
`adjacent_find()`	C++98	yes	yes		yes
`equal()`	C++98	yes	yes		
`is_permutation()`	C++11	no	yes		
`mismatch()`	C++98	yes	yes	`in_in`	yes
`lexicographical_compare()`	C++98	yes	yes		
`lexicographical_compare_three_way()`	C++20	no	no		
`is_sorted()`	C++11	yes	yes		
`is_sorted_until()`	C++11	yes	yes		yes
`is_partitioned()`	C++11	yes	yes		
`partition_point()`	C++11	no	yes		
`is_heap()`	C++11	yes	yes		
`is_heap_until()`	C++11	yes	yes		yes
`all_of()`	C++11	yes	yes		
`any_of()`	C++11	yes	yes		
`none_of()`	C++11	yes	yes		

Table 7.12. Non-modifying algorithms

Note that the algorithms `std::ranges::min()`, `std::ranges::max()`, and `std::ranges::minmax()` have only counterparts in `std` that take two values or a `std::initializer_list<>`.

Table *Modifying algorithms* lists the available modifying standard algorithms in C++20.

Name	**Since**	**Parallel**	**Ranges**	**`_result`**	**Borrowed**
`for_each()`	C++98	yes	yes	`in_fun`	yes
`for_each_n()`	C++17	yes	yes	`in_fun`	
`copy()`	C++98	yes	yes	`in_out`	yes
`copy_if()`	C++98	yes	yes	`in_out`	
`copy_n()`	C++11	yes	yes	`in_out`	yes
`copy_backward()`	C++11	no	yes	`in_out`	yes
`move()`	C++11	yes	yes	`in_out`	yes
`move_backward()`	C++11	no	yes	`in_out`	yes
`transform()` for one range	C++98	yes	yes	`in_out`	yes
`transform()` for two ranges	C++98	yes	yes	`in_in_out`	yes
`merge()`	C++98	yes	yes		
`swap_ranges()`	C++98	yes	yes	`in_in`	yes
`fill()`	C++98	yes	yes		yes
`fill_n()`	C++98	yes	yes		
`generate()`	C++98	yes	yes		yes
`generate_n()`	C++98	yes	yes		
`iota()`	C++11	no	no		
`replace()`	C++98	yes	yes		yes
`replace_if()`	C++98	yes	yes		yes
`replace_copy()`	C++98	yes	yes	`in_out`	yes
`replace_copy_if()`	C++98	yes	yes	`in_out`	yes

Table 7.13. Modifying algorithms

Table *Removing algorithms* lists the available "removing" standard algorithms in C++20. Note that algorithms never really remove elements. This still applies when they take whole ranges as single arguments. They change the order so that the elements not removed are packed to the front and return the new end.

Name	**Since**	**Parallel**	**Ranges**	**`_result`**	**Borrowed**
`remove()`	C++98	yes	yes		yes
`remove_if()`	C++98	yes	yes		yes
`remove_copy()`	C++98	yes	yes	`in_out`	yes
`remove_copy_if()`	C++98	yes	yes	`in_out`	yes
`unique()`	C++98	yes	yes		yes
`unique_copy()`	C++98	yes	yes	`in_out`	yes

Table 7.14. Removing algorithms

Table *Mutating algorithms* lists the available mutating standard algorithms in C++20.

Name	**Since**	**Parallel**	**Ranges**	**`_result`**	**Borrowed**
`reverse()`	C++98	yes	yes		yes
`reverse_copy()`	C++98	yes	yes	`in_out`	yes
`rotate()`	C++98	yes	yes		yes
`rotate_copy()`	C++98	yes	yes	`in_out`	yes
`shift_left()`	C++20	yes	no		
`shift_right()`	C++20	yes	no		
`sample()`	C++17	no	yes		
`next_permutation()`	C++98	no	yes	`in_found`	yes
`prev_permutation()`	C++98	no	yes	`in_found`	yes
`shuffle()`	C++11	no	yes		yes
`random_shuffle()`	C++98	no	no		
`partition()`	C++98	yes	yes		
`stable_partition()`	C++98	yes	yes		
`partition_copy()`	C++11	yes	yes		

Table 7.15. Mutating algorithms

Table *Sorting algorithms* lists the available sorting standard algorithms in C++20.

Name	**Since**	**Parallel**	**Ranges**	**`_result`**	**Borrowed**
`sort()`	C++98	yes	yes		yes
`stable_sort()`	C++98	yes	yes		yes
`partial_sort()`	C++98	yes	yes		yes
`partial_sort_copy()`	C++98	yes	yes	`in_out`	yes
`nth_element()`	C++98	yes	yes		yes
`partition()`	C++98	yes	yes		yes
`stable_partition()`	C++98	yes	yes		yes
`partition_copy()`	C++11	yes	yes	`in_out_out`	yes
`make_heap()`	C++98	no	yes		yes
`push_heap()`	C++98	no	yes		yes
`pop_heap()`	C++98	no	yes		yes
`sort_heap()`	C++98	no	yes		yes

Table 7.16. Sorting algorithms

Table *Algorithms for sorted ranges* lists the available standard algorithms for sorted ranges in C++20.

Name	Since	Parallel	Ranges	`_result`	Borrowed
`binary_search()`	C++98	no	yes		
`includes()`	C++98	yes	yes		
`lower_bound()`	C++98	no	yes		yes
`upper_bound()`	C++98	no	yes		yes
`equal_range()`	C++98	no	yes		yes
`merge()`	C++98	yes	yes	`in_in_out`	yes
`inplace_merge()`	C++98	yes	yes		yes
`set_union()`	C++98	yes	yes	`in_in_out`	yes
`set_intersection()`	C++98	yes	yes	`in_in_out`	yes
`set_difference()`	C++98	yes	yes	`in_out`	yes
`set_symmetric_difference()`	C++98	yes	yes	`in_in_out`	yes
`partition_point()`	C++11	no	yes		yes

Table 7.17. Algorithms for sorted ranges

Table *Numeric algorithms* lists the available numeric standard algorithms in C++20.

Name	Since	Parallel	Ranges	`_result`	Borrowed
`accumulate()`	C++98	no	no		
`reduce()`	C++17	yes	no		
`transform_reduce()`	C++17	yes	no		
`inner_product()`	C++98	no	no		
`adjacent_difference()`	C++98	no	no		
`partial_sum()`	C++98	no	no		
`inclusive_scan()`	C++17	yes	no		
`exclusive_scan()`	C++17	yes	no		
`transform_inclusive_scan()`	C++17	yes	no		
`transform_exclusive_scan()`	C++17	yes	no		
`iota()`	C++11	no	no		

Table 7.18. Numeric algorithms

This page is intentionally left blank

Chapter 8

View Types in Detail

This chapter discusses the details of all view types that the C++20 standard library introduced.

The chapter starts with an initial overview of all view types and a discussion about general aspects of views, such as their common base class provided by the C++ standard library.

Afterwards, we discuss the various view types of C++20 in multiple sections grouped by general characteristics of the views: views that are the initial building blocks of view pipelines (using existing values or generating values themselves), filtering and transforming views, mutating views, and finally, views for multiple ranges. Each description of a view type starts with an overview of the most important characteristics.

Note that the next chapter about `std::span` discusses additional details of the span view (historically, spans were not part of the ranges library, but they are also views).

8.1 Overview of All Views

C++20 provides a significant number of different views. Some of them wrap elements of existing sources, some of them generate values themselves, and many of them operate on elements or their values (filtering or transforming them).

This chapter gives a brief overview of all view types available in C++20. Note that for almost all of these types, there are auxiliary *range adaptors/factories* that allow programmers to create the views by calling a function or piping into them. In general, you should use these adaptors and factories instead of directly initializing views because the adaptors and factories perform additional optimizations (such as choosing the best among different views), work in more situations, double check requirements, and are just easier to use.

8.1.1 Overview of Wrapping and Generating Views

Table *Wrapping and generating views* lists the standard views that can only be source elements of a pipeline. These views might be

- ***Wrapping views***, which operate on a sequence of elements of an external resource (such as a container or elements read from an input stream)
- ***Factory views***, which generate elements themselves

Type	Adaptor/Factory	Effect
`std::ranges::ref_view`	`all(`*rg*`)`	Reference to a range
`std::ranges::owning_view`	`all(`*rg*`)`	View containing a range
`std::ranges::subrange`	`counted(`*beg*`,`*sz*`)`	Begin iterator and sentinel
`std::span`	`counted(`*beg*`,`*sz*`)`	Begin iterator to contiguous memory and a size
`std::ranges::iota_view`	`iota(`*val*`)` `iota(`*val*`,`*endVal*`)`	Generator of incremented values
`std::ranges::single_view`	`single(`*val*`)`	View owning exactly one value
`std::ranges::empty_view`	`empty<`*T*`>`	View with no elements
`std::ranges::basic_istream_view`		Reads elements from a stream
`std::ranges::istream_view<>`	`istream<`*T*`>(`*strm*`)`	From a `char` stream
`std::ranges::wistream_view<>`	–	From a `wchar_t` stream
`std::basic_string_view`	–	Read-only view to characters
`std::string_view`	–	To a `char` sequence
`std::u8string_view`	–	To a `char8_t` sequence
`std::u16string_view`	–	To a `char16_t` sequence
`std::u32string_view`	–	To a `char32_t` sequence
`std::wstring_view`	–	To a `wchar_t` sequence

Table 8.1. Wrapping and generating views

The table lists both the types of the views and the name of the range adaptors/factories you can use to create them (if there are any).

For string views, which were introduced with C++17 as type `std::basic_string_view<>`, C++ provides the alias types `std::string_view`, `std::u8string_view`, `std::u16string_view`, `std::u32string_view`, and `std::wstring_view`.

For istream views, which have type `std::basic_istream_view<>`, C++ provides the alias types `std::istream_view` and `std::wistream_view`.

Note that the view types use different namespaces:

- `span` and `string_view` are in the namespace `std`.
- All range adaptors and factories are provided in the namespace `std::views` (which is an alias for `std::ranges::views`).
- All other view types are provided in the namespace `std::ranges`.

In general, it is better to use the range adaptors and factories. For example, you should always favor the adaptor `std::views::all()` instead of using the types `std::ranges::ref_view<>` and `std::ranges::owning_view<>` directly. However, in some situations, there is no adaptor/factory provided, which means that you have to use the view types directly. The most important example is when you have to create a view from a pair of iterators. In that case, you have to initialize a `std::ranges::subrange` directly.

8.1.2 Overview of Adapting Views

Table *Adapting views* lists the standard views that adapt, in some way, elements of given ranges (filter out elements, modify the values of elements, change the order of elements, or combine or create sub-ranges). They can especially be used inside pipelines of views.

Type	Adaptor	Effect
`std::ranges::`**`ref_view`**	`all(`*rg*`)`	Reference to a range
`std::ranges::`**`owning_view`**	`all(`*rg*`)`	View containing a range
`std::ranges::`**`take_view`**	`take(`*num*`)`	The first (up to) *num* elements
`std::ranges::`**`take_while_view`**	`take_while(`*pred*`)`	All leading elements that match a predicate
`std::ranges::`**`drop_view`**	`drop(`*num*`)`	All except the first *num* elements
`std::ranges::`**`drop_while_view`**	`drop_while(`*pred*`)`	All except leading elements that match a predicate
`std::ranges::`**`filter_view`**	`filter(`*pred*`)`	All elements that match a predicate
`std::ranges::`**`transform_view`**	`transform(`*func*`)`	The transformed values of all elements
`std::ranges::`**`elements_view`**	`elements<`*idx*`>`	The *idx*th member/attribute of all elems
`std::ranges::`**`keys_view`**	`keys`	The first member of all elements
`std::ranges::`**`values_view`**	`values`	The second member of all elements
`std::ranges::`**`reverse_view`**	`reverse`	All elements in reverse order
`std::ranges::`**`split_view`**	`split(`*sep*`)`	All elements of a range split into sub-ranges
`std::ranges::`**`lazy_split_view`**	`lazy_split(`*sep*`)`	All elements of an input or const range split into sub-ranges
`std::ranges::`**`join_view`**	`join`	All elements of a range of multiple ranges
`std::ranges::`**`common_view`**	`common()`	All elements with same type for iterator and sentinel

Table 8.2. Adapting views

Again, you should prefer to use the range adaptors over using the view types directly. For example, you should always prefer to use the adaptor `std::views::take()` over the type `std::ranges::take_view<>`, because the adaptor might not create a take view at all when you can simply jump to the *n*-th element of the underlying range. As another example, you should always prefer to use the adaptor `std::views::common()` over the type `std::ranges::common_view<>`, because only the adaptor allows you to pass ranges that are already common (to just take them as they are). Wrapping a common range with the `common_view` results in a compile-time error.

8.2 Base Class and Namespace of Views

Views have a common type that provides most of their member functions, which is discussed in this section.

This section also discusses, why range adaptors and factories use a special namespace.

8.2.1 Base Class for Views

All standard views are derived from the class `std::ranges::`**`view_interface<`***`viewType`***`>`**.[1]

The class template `std::ranges::`**`view_interface<>`** introduces several basic member functions based on the definitions of `begin()` and `end()` of a derived view type that had to be passed to this base class as a template parameter. Table *Operations of* `std::ranges::view_interface<>` lists the API the class provides for views.

Operation	Effect	Provided if
r`.empty()`	Yields whether *r* is empty (`begin() == end()`)	At least forward iterators
`if (`*r*`)`	`true` if *r* is not empty	At least forward iterators
r`.size()`	Yields the number of elements	Can compute the difference between begin and end
r`.front()`	Yields the first element	At least forward iterators
r`.back()`	Yields the last element	At least bidirectional iterators and `end()` yields the same type as `begin()`
r`[`*idx*`]`	Yields the *n*-th element	At least random-access iterators
r`.data()`	Yields a raw pointer to the memory of the elements	Elements are in contiguous memory

Table 8.3. Operations of `std::ranges::view_interface<>`

The class `view_interface<>` also initializes `std::ranges::enable_view<>` for each type derived from it with `true`, which means that for these types, the concept `std::ranges::view` is satisfied.

Whenever you define your own view type, you should derive it from `view_interface<>` with your own type passed as an argument. For example:

```
template<typename T>
class MyView : public std::ranges::view_interface<MyView<T>> {
 public:
   ... begin() ... ;
   ... end() ... ;
   ...
};
```

[1] In the published C++20 standard, `view_interface<>` was also derived from an empty base class `std::ranges::view_base`. However, that was fixed with `http://wg21.link/lwg3549`.

Based on the return types of `begin()` and `end()`, your type then automatically provides the member functions listed in table *Operations of* `std::ranges::view_interface<>` if the preconditions for their availability fit. The `const` versions of these member functions require that the `const` version of the view type is a valid range.

C++23 will add the members `cbegin()` and `cend()`, which map to `std::ranges::cbegin()` and `std::ranges::cend()` (they were added with `http://wg21.link/p2278r4`).

8.2.2 Why Range Adaptors/Factories Have Their Own Namespace

Range adaptors and factories have their own namespace, `std::ranges::views`, for which the namespace alias `std::views` is defined:

```
namespace std {
  namespace views = ranges::views;
}
```

That way, we can use names for views that may be used in other namespaces (even in another standard namespace).

As a consequence, you can (and have to) qualify views when using them:

```
std::ranges::views::reverse    // full qualification
std::views::reverse            // shortcut
```

Usually, it is not possible to use a view without qualification. ADL does not kick in because there is no range (container or view) defined in the namespace `std::views`:

```
std::vector<int> v;
...
take(v, 3) | drop(2);   // ERROR: can't find views (may find different symbol)
```

Even views yield an object that is not part of the namespace for views (they are in `std::ranges`), which means that you still have to qualify views when using them:

```
std::vector<int> values{ 0, 1, 2, 3, 4 };
auto v1 = std::views::all(values);
auto v2 = take(v1, 3);                  // ERROR
auto v3 = std::views::take(v1, 3);      // OK
```

There is one important thing to note: **never use a `using` declaration to skip the qualification of range adaptors**:

```
using namespace std::views;             // do not do this
```

Taking the composers example, we would be in trouble if we just filtered out `values` with a local `values` object defined:

```
std::vector<int> values;
...
std::map<std::string, int> composers{ ... };
```

```
using namespace std::views;                          // do not do this

for (const auto& elem : composers | values) {  // OOPS: finds wrong values
  ...
}
```

In this example, we would use the local vector `values` instead of the view. You are lucky here, because you get a compile-time error. If you are not lucky, unqualified views find different symbols, which might have the effect that you run into undefined behavior that uses or even overwrites memory of other objects.

8.3 Source Views to External Elements

This section discusses all features of C++20 that create views that refer to existing external values (usually passed as a single range parameter, as begin iterator and a sentinel, or as a begin iterator and a count).

8.3.1 Subrange

Type:	`std::ranges::`**`subrange<>`**
Content:	All elements from passed begin to passed end
Factories:	`std::views::`**`counted()`**
	`std::views::`**`reverse()`** on reversed subrange
Element type:	Value type of passed iterators
Requires:	At least input iterators
Category:	Same as passed
Is sized range:	If common random-access iterators or a size hint are passed
Is common range:	If on common iterators
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	If passed iterators are copyable
Propagates constness:	Never

The class template `std::ranges::`**`subrange<>`** defines a view to the elements of a range that is usually passed as a pair of begin iterator and sentinel (end iterator). However, you can also pass a single range object and indirectly, a begin iterator and a count. Internally, the view itself represents the elements by storing begin (iterator) and end (sentinel).

The major use case of the subrange view is to convert a pair of begin iterator and sentinel (end iterator) into *one* object. For example:

```
std::vector<int> coll{0, 8, 15, 47, 11, -1, 13};

std::ranges::subrange s1{std::ranges::find(coll, 15),
                         std::ranges::find(coll, -1)};
print(coll);   // 15 47 11
```

You can initialize the subrange with a special sentinel for the end value:

```
std::ranges::subrange s2{coll.begin() + 1, EndValue<-1>{}};
print(s2);     // 8 15 47 11
```

Both are especially helpful for converting a pair of iterators into a range/view so that, for example, range adaptors can process the elements:

```
void foo(auto beg, auto end)
{
  // init view for all but the first five elements (if there are any):
  auto v = std::ranges::subrange{beg, end} | std::views::drop(5);
  ...
}
```

Here is a complete example that demonstrates the use of a subrange:

ranges/subrange.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>

void printPairs(auto&& rg)
{
  for (const auto& [key, val] : rg) {
    std::cout << key << ':' << val << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // English/German dictionary:
  std::unordered_multimap<std::string,std::string> dict = {
    {"strange", "fremd"},
    {"smart", "klug"},
    {"car", "Auto"},
    {"smart","raffiniert"},
    {"trait", "Merkmal"},
    {"smart", "elegant"},
  };

  // get begin and end of all elements with translation of "smart":
  auto [beg, end] = dict.equal_range("smart");

  // create subrange view to print all translations:
  printPairs(std::ranges::subrange(beg, end));
}
```

The program has the following output:

```
smart:klug smart:elegant smart:raffiniert
```

After `equal_range()` gives us a pair of the beginning and the end of all elements in `dict` that have the key `"smart"`, we use a subrange to convert these two iterators into a view (here, a view of elements that are pairs of strings):

```
std::ranges::subrange(beg, end)
```

We then can pass the view to `printPairs()` to iterate over the elements and print them out.

Note that a subrange might change its size so that elements are inserted or deleted between begin and end as long as both begin and end remain valid:

```
std::list coll{1, 2, 3, 4, 5, 6, 7, 8};

auto v1 = std::ranges::subrange(coll.begin(), coll.end());
print(v1);     // 1 2 3 4 5 6 7 8

coll.insert(++coll.begin(), 0);
coll.push_back(9);
print(v2);     // 1 0 2 3 4 5 6 7 8 9
```

Range Factories for Subranges

There is no range factory for initializing a subrange from a begin (iterator) and an end (sentinel). However, there is a range factory that might create a subrange from begin and a count:[2]

`std::views::`**`counted(`**`beg, sz`**`)`**

`std::views::counted()` creates a subrange with the first *sz* elements of the non-contiguous range, starting with the element that the iterator *beg* refers to (for a contiguous range, `counted()` creates a span view).

When `std::views::counted()` creates a subrange, the subrange gets a `std::counted_iterator` as the begin and a dummy sentinel of type `std::default_sentinel_t` as the end. This means that:

```
std::views::counted(rg.begin(), 5);
```

is equivalent to:

```
std::ranges::subrange{std::counted_iterator{rg.begin(), 5},
                      std::default_sentinel};
```

This has the effect that the count in this subrange remains stable even if elements are inserted or removed:

```
std::list coll{1, 2, 3, 4, 5, 6, 7, 8};

auto v2 = std::views::counted(coll.begin(), coll.size());
print(v2);     // 1 2 3 4 5 6 7 8

coll.insert(++coll.begin(), 0);
coll.push_back(9);
print(v2);     // 1 0 2 3 4 5 6 7
```

For more details, see the description of `std::views::counted()` and type `std::counted_iterator`.

[2] C++20 originally stated that `all()` might also yield a subrange. However, that option was removed later when owning views were introduced (see `http://wg21.link/p2415`).

Here is a complete example that demonstrates the use of a subrange via `counted()`:

ranges/subrangecounted.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>

void printPairs(auto&& rg)
{
  for (const auto& [key, val] : rg) {
    std::cout << key << ':' << val << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // English/German dictionary:
  std::unordered_multimap<std::string,std::string> dict = {
    {"strange", "fremd"},
    {"smart", "klug"},
    {"car", "Auto"},
    {"smart","raffiniert"},
    {"trait", "Merkmal"},
    {"smart", "elegant"},
  };

  printPairs(dict);
  printPairs(std::views::counted(dict.begin(), 3));
}
```

The program has the following output:

```
trait:Merkmal car:Auto smart:klug smart:elegant smart:raffiniert strange:fremd
trait:Merkmal car:Auto smart:klug
```

Note that in rare cases, std::views::reverse() might also yield a subrange. This happens when you reverse a reversed subrange. In that case, you get back the original subrange.

Special Characteristics of Subranges

A subrange may or may not be a sized range. The way this is implemented is that a subrange has a third template parameter of the enumeration type `std::ranges::subrange_kind`, which can have the values `std::ranges::unsized` or `std::ranges::sized`. That value of this template parameter can be specified or is derived from the concept `std::sized_sentinel_for` called for the iterator and sentinel type.

This has the following consequences:

- If you pass contiguous or random-access iterators of the same type, the subrange is sized:

```
std::vector vec{1, 2, 3, 4, 5};                        // has random access
...
std::ranges::subrange sv{vec.begin()+1, vec.end()-1};  // sized
std::cout << std::ranges::sized_range<decltype(sv)>;   // true
```

- If you pass iterators that do not support random access or that are not common, the subrange is not sized:

```
std::list lst{1, 2, 3, 4, 5};                          // no random access
...
std::ranges::subrange sl{++lst.begin(), --lst.end()};  // unsized
std::cout << std::ranges::sized_range<decltype(sl)>;   // false
```

- In the latter case, you can pass a size to make a subrange a sized range:

```
std::list lst{1, 2, 3, 4, 5};                          // no random access
...
std::ranges::subrange sl2{++lst.begin(), --lst.end(), lst.size()-2}; // sized
std::cout << std::ranges::sized_range<decltype(sl2)>;  // true
```

  It is undefined behavior if you initialize the view with the wrong size or use the view after elements have been inserted or deleted between the begin and end.

Interface of Subranges

Table *Operations of the class* `std::ranges::subrange<>` lists the API of a subrange.

The default constructor is provided only if the iterator is default initializable.

The constructors taking a *`szHint`* allow programmers to convert an unsized range into a sized subrange as shown above.

Iterators of a subrange refer to the underlying (or temporarily created) range. Therefore, a subrange is a borrowed range. However, note that iterators can still dangle when the underlying range is no longer there.

Tuple-Like Interface of Subranges

`std::ranges::subrange` also has a tuple-like interface to support structured bindings (introduced to C++ with C++17). Thus, you can easily initialize a begin iterator and a sentinel (end iterator) doing something like the following:

```
auto [beg, end] = std::ranges::subrange{coll};
```

For this, the class `std::ranges::subrange` provides;

- A specialization of `std::tuple_size<>`
- Specializations of `std::tuple_element<>`
- `get<>()` functions for the indexes `0` and `1`

Operation	Effect
subrange *r*{}	Creates an empty subrange
subrange *r*{*rg*}	Creates a subrange with the elements of range *rg*
subrange *r*{*rg*, *szHint*}	Creates a subrange with the elements of range *rg* specifying that the range has *szHint* elements
subrange *r*{*beg*, *end*}	Creates a subrange with the elements of range [*beg*, *end*)
subrange *r*{*beg*, *end*, *szHint*}	Creates a subrange with the elements of range [*beg*, *end*) specifying that the range has *szHint* elements
r.begin()	Yields the begin iterator
r.end()	Yields the sentinel (end iterator)
r.empty()	Yields whether *r* is empty
if (*r*)	true if *r* is not empty
r.size()	Yields the number of elements (available if sized)
r.front()	Yields the first element (available if forwarding)
r.back()	Yields the last element (available if bidirectional and common)
r[*idx*]	Yields the *n*-th element (available if random access)
r.data()	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r.next()	Yields a subrange starting with the second element
r.next(*n*)	Yields a subrange starting with the *n*-th element
r.prev()	Yields a subrange starting with the element before the first element
r.prev(*n*)	Yields a subrange starting with the *n*-th element before the first element
r.advance(*dist*)	Modifies *r* so that it starts *num* elements later (starts earlier with negative *num*)
auto [*beg*, *end*] = *r*	Initializes *beg* and *end* with begin and end/sentinel of *r*

Table 8.4. Operations of the class `std::ranges::subrange<>`

Using Subranges to Harmonize Raw Array Types

The type of subranges depends only on the type of the iterators (and whether or not `size()` is provided). This can be used to harmonize the type of raw arrays.

Consider the following example:

```
int a1[5] = {...};
int a2[10] = {...};

std::same_as<decltype(a1), decltype(a2)>            // false

std::same_as<decltype(std::ranges::subrange{a1}),
             decltype(std::ranges::subrange{a2})>   // true
```

Note that this works only for raw arrays. For `std::array<>` and other containers, iterators (may) have different types, meaning that a harmonization of the types with a subrange is at least not portable.

8.3.2 Ref View

Type:	`std::ranges::ref_view<>`
Content:	All elements of a range
Adaptors:	`std::views::all()` and all other adaptors on lvalues
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	If on sized range
Is common range:	If on common range
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	If on const-iterable range
Propagates constness:	Never

The class template `std::ranges::ref_view<>` defines a view that simply refers to a range. That way, passing the view by value has an effect like passing the range by reference.

The effect is similar to type `std::reference_wrapper<>` (introduced with C++11), which makes references to first-class objects created with `std::ref()` and `std::cref()`. However, this wrapper has the benefit that it can still be used directly as a range.

The major use case of the `ref_view` is to convert a container into a lightweight object that is cheap to copy. For example:

```
void foo(std::ranges::input_range auto coll)    // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    ...
  }
}

std::vector<std::string> coll{...};

foo(coll);                              // copies coll
foo(std::ranges::ref_view{coll});       // pass coll by reference
```

Passing a container to a coroutine, which usually has to take parameters by value, may be one application of this technique.

Note that you can only create a ref view to an lvalue (a range that has a name):

```
std::vector coll{0, 8, 15};
...
std::ranges::ref_view v1{coll};                     // OK, refers to coll
std::ranges::ref_view v2{std::move(coll)};          // ERROR
std::ranges::ref_view v3{std::vector{0, 8, 15}};    // ERROR
```

For an rvalue, you have to use an owning view.

Here is a complete example using a ref view:

ranges/refview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void printByVal(std::ranges::input_range auto coll)    // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> coll{"love", "of", "my", "life"};

  printByVal(coll);                              // copies coll
  printByVal(std::ranges::ref_view{coll});       // pass coll by reference
  printByVal(std::views::all(coll));             // ditto (using a range adaptor)
}
```

Range Adaptors for Ref Views

Ref views can also (and usually should) be created with a range factory:

`std::views::`**`all`**`(`*`rg`*`)`

`std::views::all()` creates a `ref_view` to the passed range *rg* provided *rg* is an lvalue and not a view yet (`all()` yields *rg* if it is already a view and an owning view if otherwise an rvalue is passed).

In addition, almost all other range adaptors also create a ref view if an lvalue is passed that is not a view yet.

The example program above uses this indirect way for creating a ref view with the last call of `printByVal()`. It passes a `std::ranges::ref_view<std::vector<std::string>>>`:

```
std::vector<std::string> coll{"love", "of", "my", "life"};
...
printByVal(std::views::all(coll));
```

Note that all other views that take a range indirectly call `all()` for that passed range (by using `std::views::all_t<>`). For that reason, a ref view is almost always created automatically if you pass an lvalue that is not a view to one of the views. For example, calling:

```
std::views::take(coll, 3)
```

basically has the same effect as calling the following:

```
std::ranges::take_view{std::ranges::ref_view{coll}, 3};
```

However, using the adaptor, some optimizations might apply.

For more details, see the description of `std::views::all()`.

Special Characteristics of Ref Views

The ref view stores a reference to the underlying range. Therefore, the ref view is a borrowed range. It can be used as long as the referred view is valid. For example, if the underlying view is a vector, a reallocation does ***not*** invalidate this view. However, note that iterators can still dangle when the underlying range is no longer there.

Interface of Ref Views

Table *Operations of the class* `std::ranges::ref_view<>` lists the API of a `ref_view`.

Operation	**Effect**
ref_view *r*{*rg*}	Creates a `ref_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to

Table 8.5. Operations of the class `std::ranges::ref_view<>`

Note that there is no default constructor provided for this view.

8.3.3 Owning View

Type:	`std::ranges::owning_view<>`
Content:	All elements of a moved range
Adaptors:	`std::views::all()` and all other adaptors on rvalues
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	If on sized range
Is common range:	If on common range
Is borrowed range:	If on borrowed range
Caches:	Nothing
Const iterable:	If on const-iterable range
Propagates constness:	Always

The class template `std::ranges::owning_view<>` defines a view that takes ownership of the elements of another range.[3]

This is the only view (so far) that might own multiple elements. However, construction is still cheap, because an initial range has to be an rvalue (temporary object or object marked with `std::move()`). The constructor will then move the range to an internal member of the view.

For example:

```
std::vector vec{0, 8, 15};
std::ranges::owning_view v0{vec};              // ERROR
std::ranges::owning_view v1{std::move(vec)};   // OK
print(v1);                                     // 0 8 15
print(vec);                                    // unspecified value (was moved away)

std::array<std::string, 3> arr{"tic", "tac", "toe"};
std::ranges::owning_view v2{arr};              // ERROR
std::ranges::owning_view v2{std::move(arr)};   // OK
print(v2);                                     // "tic" "tac" "toe"
print(arr);                                    // "" "" ""
```

This is the only standard view of C++20 that does not support copying at all. You can only move it. For example:

```
std::vector coll{0, 8, 15};
std::ranges::owning_view v0{std::move(coll)};

auto v3 = v0;                                  // ERROR
auto v4 = std::move(v0);                       // OK (range in v0 moved to v4)
```

[3] Originally, C++20 did not have an owning view. The view was introduced with a later fix (see `http://wg21.link/p2415`).

The major use case of the `owning_view` is to create a view from a range without depending on the lifetime of the range anymore. For example:

```
void foo(std::ranges::view auto coll)          // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    ...
  }
}

std::vector<std::string> coll{...};

foo(coll);                                     // ERROR: no view
foo(std::move(coll));                          // ERROR: no view
foo(std::ranges::owning_view{coll});           // ERROR: must pass rvalue
foo(std::ranges::owning_view{std::move(coll)}); // OK: move coll as view
```

Converting a range to an owning view is usually done implicitly by passing a temporary container to a range adaptor (see below).

Range Adaptors for Owning Views

Owning views can also (and usually should) be created with a range factory:

`std::views::`**`all`**`(`*`rg`*`)`

`std::views::all()` creates an `owning_view` to the passed range *`rg`* provided *`rg`* is an rvalue and not a view yet (`all()` yields *`rg`* if it is already a view and a ref view if otherwise an lvalue is passed).

In addition, almost all other range adaptors also create an owning view if an rvalue is passed that is not a view yet.

The example above can therefore implement the call of `foo()` as follows:

```
foo(std::views::all(std::move(coll)));         // move coll as view
```

To create an owning view, the adaptor `all()` requires an rvalue (for an lvalue, `all()` creates a ref view). This means that you have to pass a temporary range or a named range marked with `std::move()` (`all()` yields a ref view if an lvalue is passed).

Here is a complete example:

ranges/owningview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

auto getColl()
{
  return std::vector<std::string>{"you", "don't", "fool", "me"};
}
```

```
int main()
{
  std::ranges::owning_view v1 = getColl();     // view owning a vector of strings
  auto v2 = std::views::all(getColl());        // ditto
  static_assert(std::same_as<decltype(v1), decltype(v2)>);

  // iterate over drop view of owning view of vector<string>:
  for (const auto& elem : getColl() | std::views::drop(1)) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Here, `v1` and `v2` both have type `std::ranges::owning_view<std::vector<std::string>>`. However, the usual way to create an owning view is demonstrated the range-based `for` loop at the end of the program: it creates an owning view indirectly when we pass the a temporary container to a range adaptor. The expression

```
getColl() | std::views::drop(1)
```

creates a

```
std::ranges::drop_view<std::ranges::owning_view<std::vector<std::string>>>
```

The way this happens is that the drop view indirectly calls `all()` for the passed temporary range (by using `std::views::all_t<>`).

For more details, see the description of `std::views::all()`.

Special Characteristics of Owning Views

The owning view moves the passed range into itself. Therefore, the owning view is a borrowed range if the range it owns is a borrowed range.

Interface of Owning Views

Table *Operations of the class* `std::ranges::owning_view<>` lists the API of an `owning_view`.

The default constructor is provided only if the iterator is default initializable.

Operation	Effect
owning_view *r*`{}`	Creates an `owning_view` that is empty
owning_view *r*`{`*rg*`}`	Creates an `owning_view` that owns the elements of *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* owns

Table 8.6. Operations of the class `std::ranges::owning_view<>`

8.3.4 Common View

Type:	`std::ranges::`**`common_view<>`**
Content:	All elements of a range with harmonized iterator types
Adaptor:	`std::views::`**`common()`**
Element type:	Same type as passed range
Requires:	Non-common at least forward range (adaptor accepts common ranges)
Category:	Usually forward (contiguous if on a sized contiguous range)
Is sized range:	If on sized range
Is common range:	Always
Is borrowed range:	If on borrowed range
Caches:	Nothing
Const iterable:	If on const-iterable range
Propagates constness:	Only if on rvalue range

The class template `std::ranges::`**`common_view<>`** is a view that harmonizes the begin and end iterator types of a range to be able to pass them to code where the same iterator type is required (such as to constructors of containers or traditional algorithms).

Note that the constructor of the view requires that the passed range is not a common range (iterators have different types). For example:

```
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v1 = std::ranges::common_view{lst};        // ERROR: is already common

auto take5 = std::ranges::take_view{lst, 5};    // yields no common view
auto v2 = std::ranges::common_view{take5};      // OK
```

The corresponding range adaptor can also deal with ranges that already are common. Therefore, it is better to use the range adaptor `std::views::common()` to create a common view.

Range Adaptors for Common Views

Common views can also (and usually should) be created with a range adaptor:

`std::views::`**`common`**`(`*`rg`*`)`

`std::views::common()` creates a view that refers to *rg* as a common range. If *rg* is already common, it returns `std::views::all(`*`rg`*`)`.

The examples above can therefore be made to always compile as follows:

```
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v1 = std::views::common(lst);                    // OK

auto take5 = std::ranges::take_view{lst, 5};          // yields no common view
auto v2 = std::views::common(take5);                  // v2 is common view
std::vector<int> coll{v2.begin(), v2.end()};          // OK
```

Note that by using the adaptor, it is not a compile-time error to create a common view from a range that is already common. This is the key reason to prefer the adaptor over directly initializing the view.

Here is a full example program for using common views:

ranges/commonview.cpp

```cpp
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::list<std::string> coll{"You're", "my", "best", "friend"};

  auto tv = coll | std::views::take(3);
  static_assert(!std::ranges::common_range<decltype(tv)>);
  // could not initialize a container by passing begin and end of the view

  std::ranges::common_view vCommon1{tv};
  static_assert(std::ranges::common_range<decltype(vCommon1)>);

  auto tvCommon = coll | std::views::take(3) | std::views::common;
  static_assert(std::ranges::common_range<decltype(tvCommon)>);

  std::vector<std::string> coll2{tvCommon.begin(), tvCommon.end()};  // OK
  print(coll);
  print(coll2);
}
```

Here, we first create a take view, which is not common if it is not a random-access range. Because it is not common, we cannot use it to initialize a container with its elements:

```cpp
std::vector<std::string> coll2{tv.begin(), tv.end()}; // ERROR: begin/end types differ
```

Using the common view, the initialization works fine. Use `std::views::common()` if you do not know whether the range you want to harmonize is already common.

The major use case of the common view is to pass ranges or views that have different types for the begin and end iterators to generic code that requires begin and end to have the same type. A typical example is to pass the begin and end of a range to a constructor of a container or to a traditional algorithm. For example:

```
std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v1 = std::views::take(lst, 5);              // Note: types of begin() and end() differ
std::vector<int> coll{v1.begin(), v1.end()};     // ERROR: containers require the same type

auto v2 = std::views::common(std::views::take(lst, 5));    // same type now
std::vector<int> coll{v2.begin(), v2.end()};               // OK
```

Interface of Common Views

Table *Operations of the class* `std::ranges::common_view<>` lists the API of a common view.

Operation	Effect
common_view *r*{}	Creates a `common_view` that refers to a default constructed range
common_view *r*{*rg*}	Creates a `common_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to

Table 8.7. Operations of the class `std::ranges::common_view<>`

The default constructor is provided only if the iterator is default initializable.

Internally, `common_view` uses iterators of type `std::common_iterator`.

8.4 Generating Views

This section discusses all features of C++20 for creating views that generate values themselves (meaning that they do not refer to elements or values outside the view).

8.4.1 Iota View

Type:	`std::ranges::`**`iota_view<>`**
Content:	Generator of an incrementing sequence of values
Factory:	`std::views::`**`iota()`**
Element type:	Values
Category:	Input to random access (depends on the type of the start value)
Is sized range:	If initialized with an end value
Is common range:	If limited and the end has the same type as the values
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	Always
Propagates constness:	Never (but elements are not lvalues)

The class template `std::ranges::`**`iota_view<>`** is a view that generates a sequence of values. These values may be integral, such as

- 1, 2, 3 ...
- `'a'`, `'b'`, `'c'` ...

or they might use the operator ++ to generate a sequence of pointers or iterators.

The sequence might be limited or unlimited (endless).

The major use case of the iota view is to provide a view that iterates over a sequence of values. For example:

```
std::ranges::iota_view v1{1, 100};            // view with values: 1, 2, ... 99
for (auto val : v1) {
  std::cout << val << '\n';                   // print these values
}
```

Range Factories for Iota Views

Iota views can also (and usually should) be created with a range factory:

`std::views::`**`iota(`***`val`***`)`**

`std::views::`**`iota(`***`val`*, *`endVal`***`)`**

For example:

```
for (auto val : std::views::iota(1, 100)) {  // iterate over values 1, 2, ... 99
  std::cout << val << '\n';                   // print these values
}
```

Here is a full example program using iota views:

ranges/iotaview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  int num = 0;
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
    if (++num > 30) {   // print only up to 30 values:
      std::cout << "...";
      break;
    }
  }
  std::cout << '\n';
}

int main()
{
  std::ranges::iota_view<int> iv0;            // 0 1 2 3 ...
  print(iv0);

  std::ranges::iota_view iv1{-2};             // -2 -1 0 1 ...
  print(iv1);

  std::ranges::iota_view iv2{10, 20};         // 10 11 12 ... 19
  print(iv2);

  auto iv3 = std::views::iota(1);             // -2 -1 0 1 ...
  print(iv3);

  auto iv4 = std::views::iota('a', 'z'+1);    // a b c ... z
  print(iv4);

  std::vector coll{0, 8, 15, 47, 11};
  for (auto p : std::views::iota(coll.begin(), coll.end())) { // sequence of iterators
    std::cout << *p << ' ';                   // 0 8 15 47 11
  }
  std::cout << '\n';
}
```

The program has the following output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ...
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 ...
10 11 12 13 14 15 16 17 18 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 ...
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 8 15 47 11
```

Note that the output from `'a'` to `'z'` depends on the character set of the platform.

Special Characteristics of Iota Views

Because the iterators hold the current value, the iota view is a borrowed range (the lifetime of its iterators does not depend on the view). If the iota view is limited and the end value has the same type as the begin value, the view is a common range.

If the iota view is endless or the type of the end differs from the type of the begin, it is not a common range. You might have to use a common view to harmonize the types of the begin iterator and the sentinel (end iterator).

Interface of Iota Views

Table *Operations of the class* `std::ranges::iota_view<>` lists the API of an iota view.

Operation	Effect
iota_view<*Type*> *v*	Creates an unlimited sequence starting with the default value of *Type*
iota_view *v{begVal}*	Creates an unlimited sequence starting with *begVal*
iota_view *v{begVal, endVal}*	Creates a sequence starting with *begVal* up to the value before *endVal*
iota_view *v{beg, end}*	Helper constructor to enable sub-views
v`.begin()`	Yields the begin iterator (referring to the starting value)
v`.end()`	Yields the sentinel (end iterator), which is `std::unreachable_sentinel` if unlimited
v`.empty()`	Yields whether *v* is empty
`if (`*v*`)`	`true` if *v* is not empty
v`.size()`	Yields the number of elements (available if limited and computable)
v`.front()`	Yields the first element
v`.back()`	Yields the last element (available if limited and common)
v`[`*idx*`]`	Yields the *n*-th element

Table 8.8. Operations of the class `std::ranges::iota_view<>`

When declaring an iota view as follows:

```
std::ranges::iota_view v1{1, 100};        // range with values: 1, 2, ... 99
```

the type of v1 is deduced as std::ranges::iota_view<int, int>, which creates an object that provides the basic API, with begin() and end() yielding an iterator to these values. The iterators store the current value internally and increment it when the iterator is incremented:

```
std::ranges::iota_view v1{1, 100};        // range with values: 1, 2, ... 99
auto pos = v1.begin();                    // initialize iterator with value 1
std::cout << *pos;                        // print current value (here, value 1)
++pos;                                    // increment to next value (i.e., increment value to 2)
std::cout << *pos;                        // print current value (here, value 2)
```

Thus, the type of the values has to support the operator ++. For this reason, a value type such as bool or std::string is not possible.

Note that the type of this iterator is up to the implementation of iota_view, which means that you have to use auto when using it. Alternatively, you could declare pos with std::ranges::iterator_t<>:

```
std::ranges::iterator_t<decltype(v1)> pos = v1.begin();
```

As for ranges of iterators, the range of values is a half-open range and the end value is not included. To include the end value, you might have to increment the end :

```
std::ranges::iota_view letters{'a', 'z'+1};   // values: 'a', 'b', ... 'z'
for (auto c : letters) {
  std::cout << c << ' ';                      // print these values/letters
}
```

Note that the output of this loop depends on the character set. Only if the lower-case letters have consecutive values (as is the case with ASCII or ISO-Latin-1 or UTF-8) the view iterates over nothing else but lower-case characters. By using char8_t, you can ensure portably that this is the case for UTF-8 characters:

```
std::ranges::iota_view letters{u8'a', u8'z'+1};   // UTF-8 values from a to z
```

If no end value is passed, the view is unlimited and generates an endless sequence of values:

```
std::ranges::iota_view v2{10L};           // unlimited range with values: 10L, 11L, 12L, ...
std::ranges::iota_view<int> v3;           // unlimited range with values: 0, 1, 2, ...
```

The type of v3 is std::ranges::iota_view<int, std::unreachable_sentinel_t>, meaning that end() yields std::unreachable_sentinel.

An unlimited iota view is endless. When the iterator represents the highest value and iterates to the next value, it calls the operator ++, which formally, is undefined behavior (in practice, it usually performs an overflow, so that the next value is the lowest value of the value type). If the view has an end value that never matches, it behaves the same.

Using Iota Views to Iterate over Pointers and Iterators

You can initialize an iota view with iterators or pointers. In that case, the iota view uses the first value as begin and the last value as end, but the elements are iterators/pointers instead of the values.

You can use this to deal with iterators to all elements of a range.

```
std::list<int> coll{2, 4, 6, 8, 10, 12, 14};
...
// pass iterator to each element to foo():
```

```
for (const auto& itorElem : std::views::iota(coll.begin(), coll.end())) {
  std::cout << *itorElem << '\n';
}
```

You can also use this feature to initialize a container with ***iterators*** to all elements of a collection `coll`:

```
std::ranges::iota_view itors{coll.begin(), coll.end()};
std::vector<std::ranges::iterator_t<decltype(coll)>> refColl{itors.begin(),
                                                             itors.end()};
```

Note that if you skip the specification of the element type of `refColl`, you have to use parentheses. Otherwise, you would initialize the vector with two iterator elements:

```
std::vector refColl(itors.begin(), itors.end());
```

The main reason that this constructor is provided is to support generic code that can create a sub-view of an iota view, like, for example, the drop view does:[4]

```
// generic function to drop the first element from a view:
auto dropFirst = [] (auto v) {
                   return decltype(v){++v.begin(), v.end()};
                 };

std::ranges::iota_view v1{1, 9};    // iota view with elems from 1 to 8
auto v2 = dropFirst(v1);            // iota view with elems from 2 to 8
```

In this case, the availability of the member functions `size()` and operator`[]` depends on the support for these operators in the passed range.

[4] Thanks to Hannes Hauswedell for pointing this out.

8.4.2 Single View

Type:	`std::ranges::single_view<>`
Content:	Generator of a range with a single element
Factory:	`std::views::single()`
Element type:	Reference
Category:	Contiguous
Is sized range:	Always with size 1
Is common range:	Always
Is borrowed range:	Never
Caches:	Nothing
Const iterable:	Always
Propagates constness:	Always

The class template `std::ranges::single_view<>` is a view that owns one element. Unless the value type is `const`, you can even modify the value.

The overall effect is that a single view behaves roughly like a cheap collection of one element that does not allocate any heap memory.

The major use case of the single view is to call generic code to operate on a cheap view with exactly one element/value:

```
std::ranges::single_view<int> v1{42};         // single view
for (auto val : v1) {
  ...                                          // called once
}
```

This can be used for test cases or for cases where code has to provide a view that in a specific context, has to contain exactly one element.

Range Factories for Single Views

Single views can also (and usually should) be created with a range factory:

```
    std::views::single(val)
```

For example:

```
for (auto val : std::views::single(42)) {     // iterate over the single int value 42
  ...                                          // called once
}
```

Here is a full example program using single views:

ranges/singleview.cpp

```
#include <iostream>
#include <string>
#include <ranges>
```

```
void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::ranges::single_view<double> sv0;               // single view double 0.0
  std::ranges::single_view sv1{42};                   // single_view<int> with int 42

  auto sv2 = std::views::single('x');                 // single_view<char>
  auto sv3 = std::views::single("ok");                // single_view<const char*>
  std::ranges::single_view<std::string> sv4{"ok"};   // single view with string "ok"

  print(sv0);
  print(sv1);
  print(sv2);
  print(sv3);
  print(sv4);
  print(std::ranges::single_view{42});
}
```

The program has the following output:

```
0
42
x
ok
ok
42
```

Special Characteristics of Single Views

The single view models the following concepts:

- `std::ranges::contiguous_range`
- `std::ranges::sized_range`

The view is always a common range and never a borrowed range (the lifetime of its iterators does not depend on the view).

Interface of Single Views

Table *Operations of the class* `std::ranges::single_view<>` lists the API of a single view.

Operation	Effect
single_view`<`*`Type`*`>` *v*	Creates a view with one value-initialized element of type *`Type`*
single_view *v*`{`*`val`*`}`	Creates a view with one element of value *`val`* of its type
single_view`<`*`Type`*`>` *v*`{std::in_place,`*`arg1`*`,`*`arg2`*`,...}`	Creates a view with one element initialized with *`arg1`*, *`arg2`*, ...
v`.begin()`	Yields a raw pointer to the element
v`.end()`	Yields a raw pointer to the position behind the element
v`.empty()`	Always `false`
`if (`*v*`)`	Always `true`
v`.size()`	Yields `1`
v`.front()`	Yields a reference to the current value
v`.back()`	Yields a reference to the current value
v`[`*`idx`*`]`	Yields the current value for *`idx`* `0`
r`.data()`	Yields a raw pointer to the element

Table 8.9. Operations of the class `std::ranges::single_view<>`

If you call the constructor without passing an initial value, the element in the single view is *value-initialized*. That means it either uses the default constructor of its *`Type`* or initializes the value with `0`, `false`, or `nullptr`.

To initialize a single view with objects that need multiple initializers, you have two options:

- Passing the initialized object:

  ```
  std::ranges::single_view sv1{std::complex{1,1}};
  auto sv2 = std::views::single(std::complex{1,1});
  ```

- Passing the initial values as arguments after an argument `std::in_place`:

  ```
  std::ranges::single_view<std::complex<int>> sv4{std::in_place, 1, 1};
  ```

Note that for a non-const single view, you can modify the value of the “element:”

```
std::ranges::single_view v2{42};
std::cout << v2.front() << '\n';          // prints 42
++v2.front();                             // OK, modifies value of the view
std::cout << v2.front() << '\n';          // prints 43
```

You can prevent this by declaring the element type as `const`:

```
std::ranges::single_view<const int> v3{42};
++v3.front();                             // ERROR
```

The constructor taking a value either copies or moves this values into the view. This means that the view cannot refer to the initial value (declaring the element type to be a reference does not compile).

Because `begin()` and `end()` yield the same value, the view is always a common range. Because iterators refer to the value stored in the view to be able to modify it, the single view is *not* a borrowed range (the lifetime of its iterators *does* depend on the view).

8.4.3 Empty View

Type:	`std::ranges::`**`empty_view<>`**
Content:	Generator of a range with no element
Factory:	`std::views::`**`empty<>`**
Element type:	Reference
Category:	Contiguous
Is sized range:	Always with size 0
Is common range:	Always
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	Always
Propagates constness:	Never

The class template `std::ranges::`**`empty_view<>`** is a view with no elements. However, you have to specify the element type.

The major use case of the empty view is to call generic code with a cheap view that has no elements and for which the type system knows that it never has any elements:

```
std::ranges::empty_view<int> v1;             // empty view
for (auto val : v1) {
  ...                                         // never called
}
```

This can be used for test cases or for cases where code has to provide a view that in a specific context, can never have any elements.

Range Factories for Empty Views

Empty views can also (and usually should) be created with a range factory, which which is a variable template, meaning that you declare it with a template parameter for the type but no call parameters:

`std::views::`**`empty`**`<`*`type`*`>`

For example:

```
for (auto val : std::views::empty<int>) {    // iterate over no int values
  ...                                         // never called
}
```

Here is a full example program using empty views:

ranges/emptyview.cpp

```
#include <iostream>
#include <string>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  if (coll.empty()) {
    std::cout << "<empty>\n";
  }
  else {
    for (const auto& elem : coll) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
}

int main()
{
  std::ranges::empty_view<double> ev0;     // empty view of double
  auto ev1 = std::views::empty<int>;       // empty view of int

  print(ev0);
  print(ev1);
}
```

The program has the following output:

```
<empty>
<empty>
```

Special Characteristics of Empty Views

An empty view behaves roughly like an empty vector. In fact, it models the following concepts:

- `std::ranges::contiguous_range` (which implies `std::ranges::random_access_range`)
- `std::ranges::sized_range`

Both `begin()` and `end()` simply always yield the `nullptr`, meaning that the range is also a common range and a borrowed range (the lifetime of its iterators does not depend on the view).

Interface of Empty Views

Table *Operations of the class* `std::ranges::empty_view<>` lists the API of an empty view.

You might wonder why an empty view provides `front()`, `back()`, and `operator[]` that ***always*** have undefined behavior. Calling them is ***always*** a fatal runtime error. However, remember that generic code always has to check (or know) that there are elements before calling `front()`, `back()`, or the operator `[]`, meaning that this generic code would never call these member functions. Such code will compile even for empty views.

Operation	Effect
empty_view<*Type*> *v*	Creates a view with no elements of type *Type*
v.begin()	Yields a raw pointer to the element type initialized with the nullptr
v.end()	Yields a raw pointer to the element type initialized with the nullptr
v.empty()	Yields true
if (*v*)	Always false
v.size()	Yields 0
v.front()	Always **undefined behavior** (fatal runtime error)
v.back()	Always **undefined behavior** (fatal runtime error)
v[*idx*]	Always **undefined behavior** (fatal runtime error)
r.data()	Yields a raw pointer to the element type initialized with the nullptr

Table 8.10. Operations of the class std::ranges::empty_view<>

For example, you can pass an empty view to code like this and it will compile:

```
void foo(std::ranges::random_access_range auto&& rg)
{
  std::cout << "sortFirstLast(): \n";
  std::ranges::sort(rg);
  if (!std::ranges::empty(rg)) {
    std::cout << "  first: " << rg.front() << '\n';
    std::cout << "  last:  " << rg.back() << '\n';
  }
}

foo(std::ranges::empty_view<int>{});   // OK
```

8.4.4 IStream View

Type:	`std::ranges::`**`basic_istream_view<>`**
	`std::ranges::`**`istream_view<>`**
	`std::ranges::`**`wistream_view<>`**

Content:	Generator of a range with elements read from a stream
Factory:	`std::views::`**`istream<>()`**
Element type:	Reference
Category:	Input
Is sized range:	Never
Is common range:	Never
Is borrowed range:	Never
Caches:	Nothing
Const iterable:	Never
Propagates constness:	—

The class template `std::ranges::`**`basic_istream_view<>`** is a view that reads elements from an input stream (such as the standard input, from a file, or from a string stream).

As usual for stream types, the type is generic for the type of the characters and provides specializations for `char` and `wchar_t`:

- The class template `std::ranges::`**`istream_view<>`** is a view that reads elements from an input stream using characters of type `char`.
- The class template `std::ranges::`**`wistream_view<>`** is a view that reads elements from an input stream using characters of type `wchar_t`.

For example:

```
std::istringstream myStrm{"0 1 2 3 4"};

for (const auto& elem : std::ranges::istream_view<int>{myStrm}) {
  std::cout << elem << '\n';
}
```

Range Factories for IStream Views

Istream views can also (and usually should) be created with a range factory. The factory passes its parameters to the `std::ranges::basic_istream_view` constructor using the character type of the passed range:

`std::views::`**`istream<`***`Type`***`>(`***`rg`***`)`**

For example:

```
std::istringstream myStrm{"0 1 2 3 4"};

for (const auto& elem : std::views::istream<int>(myStrm)) {
  std::cout << elem << '\n';
}
```

or:

```
std::wistringstream mywstream{L"1.1 2.2 3.3 4.4"};
auto vw = std::views::istream<double>(mywstream);
```

The initialization of vw is equivalent to both of the following initializations:

```
auto vw2 = std::ranges::basic_istream_view<double, wchar_t>{myStrm};
auto vw3 = std::ranges::wistream_view<double>{myStrm};
```

Here is a full example program using istream views:

ranges/istreamview.cpp

```
#include <iostream>
#include <string>
#include <sstream>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << "  ";
  }
  std::cout << '\n';
}

int main()
{
  std::string s{"2 4 6 8 Motorway 1977 by Tom Robinson"};

  std::istringstream mystream1{s};
  std::ranges::istream_view<std::string> vs{mystream1};
  print(vs);

  std::istringstream mystream2{s};
  auto vi = std::views::istream<int>(mystream2);
  print(vi);
}
```

The program has the following output:

```
2  4  6  8  Motorway  1977  by  Tom  Robinson
2  4  6  8
```

The program iterates twice over an string stream initialized with the string s:

- The istream view vs iterates over all strings read from the input string stream mystream1. It prints all sub-strings of s.
- The istream view vi iterates over all ints read from the input string stream mystream2. Its reading ends when it reaches Motorway.

IStream Views and `const`

Note that you ***cannot*** iterate over a `const` istream view. For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::istringstream myStrm{"0 1 2 3 4"};

printElems(std::views::istream<int>(myStrm));   // ERROR
```

The problem is that `begin()` is provided only for a non-`const` istream view because a value is processed in two steps (`begin()`/`++` and then `operator*`) and the value is stored in the view in the meantime.

The reason for modifying the view is demonstrated by the following example of reading strings from an istream view by using a low-level way to iterate over the "elements" of the view:

```
std::istringstream myStrm{"stream with very-very-very-long words"};

auto v = std::views::istream<std::string>(myStrm);

for (auto pos = v.begin(); pos != v.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

This code has the following output:

```
stream
with
very-very-very-long
words
```

When the iterator `pos` iterates over the view `va`, it reads strings from the input stream `myStrm` with `begin()` and `++`. These strings have to be stored internally so that they can be used via the operator `*`. If we store the string in the iterator, the iterator might have to hold memory for the string and copying the iterator would have to copy this memory. However, copying an iterator should not be expensive. Therefore, the view stores the string in the view, which has the effect that the view is modified while we are iterating.

As a consequence, you have to use universal/forwarding references to support this view in generic code:

```
void printElems(auto&& coll) {
  ...
}
```

Interface of IStream Views

Table *Operations of the class* `std::ranges::basic_istream_view<>` lists the API of an istream view.

Operation	Effect
istream_view *<Type> v{strm}*	Creates an istream view of type *Type* reading from *strm*
v`.begin()`	Reads the first value and yields the begin iterator
v`.end()`	Yields `std::default_sentinel` as end iterator

Table 8.11. Operations of the class `std::ranges::basic_istream_view<>`

All you can do is to use iterators to read value by value until the end of the stream is reached. No other member function (like `empty()` or `front()`) is provided.

Note that the original specification of istream views in the C++20 standard had some inconsistencies with other views, which were fixed afterwards (see `http://wg21.link/p2432`). With this fix, we have the current behavior:

- You can fully specify all template parameters:
```
std::ranges::basic_istream_view<int, char, std::char_traits<char>> v1{myStrm};
```
- The last template parameter has a working default value, meaning that you can use:
```
std::ranges::basic_istream_view<int, char> v2{myStrm};   // OK
```
- However, you cannot skip more parameters:
```
std::ranges::basic_istream_view<int> v3{myStrm};         // ERROR
```
- However, the istream views follow the usual convention that there are special types for `char` and `wchar_t` streams without the `basic_` prefix:
```
std::ranges::istream_view<int> v4{myStrm};               // OK for char streams

std::wistringstream mywstream{L"0 1 2 3 4"};
std::ranges::wistream_view<int> v5{mywstream};           // OK for wchar_t streams
```
- And a corresponding range factory is provided:
```
auto v6 = std::views::istream<int>(myStrm);              // OK
```

8.4.5 String View

Type:	std::**basic_string_view<>** std::**string_view** std::**u8string_view** std::**u16string_view** std::**u32string_view** std::**wstring_view**
Content:	All characters of a character sequence
Factory:	–
Element type:	const reference
Category:	Contiguous
Is sized range:	Always
Is common range:	Always
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	Always
Propagates constness:	Elements are always const

The class template std::**basic_string_view<>** and its specializations std::**string_view**, std::**u16string_view**, std::**u32string_view**, and std::**wstring_view** are the only view types that were already available in the C++17 standard library. However, C++20 adds the specialization for UTF-8 characters: std::**u8string_view**.

String views do not follow a couple of conventions that views usually have:

- The view is defined in the namespace std (instead of std::ranges).
- The view has its own header file <string_view>.
- The view has no range adaptor/factory to create view objects.
- The view provides only read access to the elements/characters.
- The view provides cbegin() and cend() member functions.
- The view supports no conversion to bool.

For example:

```
for (char c : std::string_view{"hello"}) {
  std::cout << c << ' ';
}
std::cout << '\n';
```

This loop has the following output:

```
h e l l o
```

Special Characteristics of String Views

Iterators do not refer to this view. Instead, they refer to the underlying character sequence. Therefore, a string view is a borrowed range. However, note that iterators can still dangle when the underlying character sequence is no longer there.

View-Specific Interface of String Views

Table *View operations of the class* `std::basic_string_view<>` lists the view-relevant parts of the API of a string view.

Operation	**Effect**
string_view *sv*	Creates a string view with no elements
string_view *sv*{*s*}	Creates a string view to *s*
v`.begin()`	Yields a raw pointer to the element type initialized with the `nullptr`
v`.end()`	Yields a raw pointer to the element type initialized with the `nullptr`
sv`.empty()`	Yields whether *sv* is empty
sv`.size()`	Yields the number of characters
sv`.front()`	Yields the first character
sv`.back()`	Yields the last character
sv`[`*idx*`]`	Yields the *n*-th character
sv`.data()`	Yields a raw pointer to the characters or `nullptr`
...	Several other operations provided for read-only strings

Table 8.12. View operations of the class `std::ranges::basic_string_view<>`

In addition to the usual interface of views, the type also provides the full API of all read-only operations of strings.

For more details, take a look at my book *C++17 - The Complete Guide* (see `http://www.cppstd17.com`).

8.4.6 Span

Type:	`std::`**`span<>`**
Content:	All elements of a range in contiguous memory
Factory:	`std::views::`**`counted()`**
Element type:	Reference
Requires:	Contiguous range
Category:	Contiguous
Is sized range:	Always
Is common range:	Always
Is borrowed range:	Always
Caches:	Nothing
Const iterable:	Always
Propagates constness:	Never

The class template `std::`**`span<>`** is a view to a sequence of elements that are stored in contiguous memory. It is described in its own chapter in detail. Here, we present only the properties of spans as a view.

The major benefit of spans is that they support referring to a subrange of *n* elements in the middle or at the end of the referred range. For example:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

// sort the three elements in the middle:
std::ranges::sort(std::span{vec}.subspan(1, 3));

// print last three elements:
print(std::span{vec}.last(3));
```

This example is discussed later in a section of the chapter about spans.

Note that spans do not follow a couple of conventions that views usually have:

- The view is defined in the namespace `std` instead of `std::ranges`.
- The view has its own header file `<span>`.
- The view supports no conversion to `bool`.

Range Factories for Spans

There is no range factory for initializing a span from a begin (iterator) and an end (sentinel). However, there is a range factory that might create a span from begin and a count:

`std::views::`**`counted(`***`beg`*, *`sz`***`)`**

`std::views::counted()` creates a span of dynamic size initialized with the first *`sz`* elements of the contiguous range, starting with the iterator *`beg`* (for a non-contiguous range, `counted()` creates a subrange).

For example:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::views::counted(vec.begin()+1, 3);    // span with 2nd to 4th elem of vec
```

When `std::views::counted()` creates a span, the span has dynamic extent (its size can vary).

Special Characteristics of Spans

Iterators do not refer to a span. Instead, they refer to the underlying elements. Therefore, a span is a borrowed range. However, note that iterators can still dangle when the underlying character sequence is no longer there.

View-Specific Interface of Spans

Table *View operations of the class* `std::span<>` lists the view-relevant parts of the API of a span.

Operation	Effect
span *sp*	Creates a span with no elements
span *sp*{*s*}	Creates a span to *s*
sp.begin()	Yields the begin iterator
sp.end()	Yields the sentinel (end iterator)
sp.empty()	Yields whether *sv* is empty
sp.size()	Yields the number of characters
sp.front()	Yields the first character
sp.back()	Yields the last character
sp[*idx*]	Yields the *n*-th character
sp.data()	Yields a raw pointer to the characters or `nullptr`
...	See the section about spans operations

Table 8.13. View operations of the class `std::span<>`

For more details, see the section about span operations.

8.5 Filtering Views

This section discusses all views that filter out elements of a given range or view.

8.5.1 Take View

Type:	`std::ranges::take_view<>`
Content:	The first (up to) *num* elements of a range
Adaptor:	`std::views::take()`
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	If on a sized range
Is common range:	If on sized random-access range
Is borrowed range:	If on borrowed view or on lvalue non-view
Caches:	Nothing
Const iterable:	If on const-iterable range
Propagates constness:	Only if on rvalue range

The class template `std::ranges::take_view<>` defines a view that refers to the first *num* elements of a passed range. If the passed range does not have enough elements, the view refers to all elements.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2 3 4 5
```

Range Adaptors for Take Views

Take views can also (and usually should) be created with a range adaptor:

`std::views::take(`*rg*`,` *n*`)`

For example:

```
for (const auto& elem : std::views::take(rg, 5)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::take(5)) {
  std::cout << elem << ' ';
}
```

Note that the adaptor might not always yield a `take_view`:

- If an `empty_view` is passed, that view is just returned.
- If a sized random-access range is passed, where you can just initialize a range of the same type with the begin being the `begin() +` *num*, such a range is returned (this applies to `subrange`, iota view, string view, and span).

For example:

```
std::vector<int> vec;

// using constructors:
std::ranges::take_view tv1{vec, 5};                    // take view of ref view of vector
std::ranges::take_view tv2{std::vector<int>{}, 5};     // take view of owing view of vector
std::ranges::take_view tv3{std::views::iota(1,9), 5};    // take view of iota view

// using adaptors:
auto tv4 = std::views::take(vec, 5);                   // take view of ref view of vector
auto tv5 = std::views::take(std::vector<int>{}, 5);   // take view of owning view of vector
auto tv6 = std::views::take(std::views::iota(1,9), 5);  // pure iota view
```

The take view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The iterators are the iterators of the passed range if it is a sized random-access range or a counted iterator to it and a default sentinel. Therefore, the range is common only if a sized random-access range is passed.

Here is a full example program using take views:

ranges/takeview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                          // 1 2 3 4 1 2 3 4 1
  print(std::ranges::take_view{coll, 5});  //1 2 3 4 1
  print(coll | std::views::take(5));       //1 2 3 4 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
1 2 3 4 1
1 2 3 4 1
```

Special Characteristics of Take Views

Note that a take view is common (has same type for iterator and sentinel) only if the underlying range is a sized range and a common range. To harmonize types, you might have to use a common view.

Interface of Take Views

Table *Operations of the class* `std::ranges::take_view<>` lists the API of a take view.

Operation	Effect
take_view *r*{}	Creates a `take_view` that refers to a default constructed range
take_view *r*{*rg*, *num*}	Creates a `take_view` that refers to the first *num* elements of range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.14. Operations of the class `std::ranges::take_view<>`

8.5.2 Take-While View

Type:	`std::ranges::take_while_view<>`
Content:	All leading elements of a range that match a predicate
Adaptor:	`std::views::take_while()`
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	Never
Is common range:	Never
Is borrowed range:	Never
Caches:	Nothing
Const iterable:	If on const-iterable range and predicate works on const values
Propagates constness:	Only if on rvalue range

The class template `std::ranges::`**`take_while_view<>`** defines a view that refers to all leading elements of a passed range that match a certain predicate. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_while_view{rg, [](auto x) {
                                                           return x % 3 != 0;
                                                         }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2
```

Range Adaptors for Take-While Views

Take-while views can also (and usually should) be created with a range adaptor. The adaptor simply passes its parameters to the `std::ranges::take_while_view` constructor:

`std::views::`**`take_while(`**_`rg`_`, `_`pred`_**`)`**

For example:

```
for (const auto& elem : std::views::take_while(rg, [](auto x) {
                                                     return x % 3 != 0;
                                                   })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::take_while([](auto x) {
                                                      return x % 3 != 0;
                                                    })) {
  std::cout << elem << ' ';
}
```

The passed predicate must be a callable that satisfies the concept std::predicate. This implies the concept std::regular_invocable, which means that the predicate should never modify the passed value of the underlying range. However, not modifying the value is a semantic constraint and this cannot always be checked at compile time. As a consequence, the predicate should at least be declared to take the argument by value or by const reference.

The take-while view stores the passed range internally (optionally converted to a view in the same way as with all()). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The iterators are just the iterators of the passed range and a special internal sentinel type.

Here is a full example program using take-while views:

ranges/takewhileview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                                    //1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::take_while_view{coll, less4}); //1 2 3
  print(coll | std::views::take_while(less4));      //1 2 3
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
1 2 3
1 2 3
```

Interface of Take-While Views

Table *Operations of the class* `std::ranges::take_while_view<>` lists the API of a take-while view.

Operation	**Effect**
take_while_view *r*`{}`	Creates a `take_while_view` that refers to a default constructed range
take_while_view *r*`{`*rg*`,` *pred*`}`	Creates a `take_while_view` that refers to the leading elements of range *rg* for which *pred* is `true`
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns
r`.pred()`	Yields a reference to the predicate

Table 8.15. Operations of the class `std::ranges::take_while_view<>`

8.5.3 Drop View

Type:	`std::ranges::drop_view<>`
Content:	All but the first *num* elements of a range
Adaptor:	`std::views::drop()`
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	If on a sized range
Is common range:	If on common range
Is borrowed range:	If on borrowed view or on lvalue non-view
Caches:	Caches `begin()` if no random-access range or no sized range
Const iterable:	If on const-iterable range that provides random access and is sized
Propagates constness:	Only if on rvalue range

The class template `std::ranges::drop_view<>` defines a view that refers to all but the first *num* elements of a passed range. It yields the opposite elements to the take view.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::drop_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
6 7 8 9 10 11 12 13
```

Range Adaptors for Drop Views

Drop views can also (and usually should) be created with a range adaptor:

`std::views::drop(rg, n)`

For example:

```
for (const auto& elem : std::views::drop(rg, 5)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::drop(5)) {
  std::cout << elem << ' ';
}
```

Note that the adaptor might not always yield a `drop_view`:

- If an empty view is passed, that view is just returned.

- If a sized random-access range is passed, where you can just initialize a range of the same type with the begin being the `begin() + ` *num*, such a range is returned (this applies to `subrange`, iota view, string view, and span).

For example:

```
std::vector<int> vec;

// using constructors:
std::ranges::drop_view dv1{vec, 5};                      // drop view of ref view of vector
std::ranges::drop_view dv2{std::vector<int>{}, 5};   // drop view of owing view of vector
std::ranges::drop_view dv3{std::views::iota(1,10), 5};    // drop view of iota view

// using adaptors:
auto dv4 = std::views::drop(vec, 5);                     // drop view of ref view of vector
auto dv5 = std::views::drop(std::vector<int>{}, 5); // drop view of owing view of vector
auto dv6 = std::views::drop(std::views::iota(1,10), 5);    // pure iota view
```

The drop view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The begin iterator is initialized and cached with the first call of `begin()`. On a range without random access this takes linear time. Therefore, it is better to reuse a drop view than to create it again from scratch.

Here is a full example program using drop views:

ranges/dropview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                               //1 2 3 4 1 2 3 4 1
  print(std::ranges::drop_view{coll, 5}); //2 3 4 1
  print(coll | std::views::drop(5));        //2 3 4 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
2 3 4 1
2 3 4 1
```

Drop Views and Caching

For better performance (to get an amortized constant complexity), drop views cache the result of `begin()` in the view (unless the range is only an input range). This means that the first iteration over the elements of a drop view is more expensive than further iterations.

For this reason, it is better to initialize a drop view once and use it twice:

```
// better:
auto v1 = coll | std::views::drop(5);
check(v1);
process(v1);
```

than to initialize and use it twice:

```
// worse:
check(coll | std::views::drop(5));
process(coll | std::views::drop(5));
```

Note that for ranges that have random access (e.g., arrays, vectors, and deques), the cached offset for the beginning is copied with the view. Otherwise, the cached beginning is not copied.

This caching may have functional consequences when the range that a filter view refers to is modified. For example:

ranges/dropcache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list   lst{1, 2, 3, 4};
  auto vVec = vec | std::views::drop(2);
  auto vLst = lst | std::views::drop(2);

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);

  print(vVec);     // OK: 2 3 4
  print(vLst);     // OK: 2 3 4

  // insert more elements at the front (=> 98 99 0 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {98, 99, 0, -1});
  lst.insert(lst.begin(), {98, 99, 0, -1});

  print(vVec);     // OK: 0 -1 0 1 2 3 4
  print(vLst);     // OOPS: 2 3 4

  // creating a copy heals:
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2);    // OK: 0 -1 0 1 2 3 4
  print(vLst2);    // OK: 0 -1 0 1 2 3 4
}
```

Note that formally, the copy of the invalid view to the vector creates undefined behavior because the C++ standard does not specify how the caching is done. Because reallocation of a vector invalidates all iterators, a cached iterator would become invalid. However, for random-access ranges, the view usually caches an offset, not the iterator. This means that the view is still valid in that it still contains the range without the first two elements.

As a rule of thumb, **do not use a drop view for which `begin()` has been called after the underlying range has been modified**.

Drop View and `const`

Note that you ***cannot always*** iterate over a `const` drop view. In fact, the referenced range has to be a random-access range and a sized range.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}
```

```
std::vector vec{1, 2, 3, 4, 5};
std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop(3));    // OK
printElems(lst | std::views::drop(3));    // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& coll) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(lst | std::views::drop(3));    // OK
```

Interface of Drop Views

Table *Operations of the class* `std::ranges::drop_view<>` lists the API of a drop view.

Operation	Effect
drop_view *r*{}	Creates a `drop_view` that refers to a default constructed range
drop_view *r*{*rg*, *num*}	Creates a `drop_view` that refers to all but the first *num* elements of range *rg*
r.`begin()`	Yields the begin iterator
r.`end()`	Yields the sentinel (end iterator)
r.`empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r.`size()`	Yields the number of elements (available if it refers to a sized range)
r.`front()`	Yields the first element (available if forwarding)
r.`back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r.`data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r.`base()`	Yields a reference to the range that *r* refers to or owns

Table 8.16. Operations of the class `std::ranges::drop_view<>`

8.5.4 Drop-While View

Type:	`std::ranges::`**`drop_while_view<>`**
Content:	All but the leading elements of a range that match a predicate
Adaptor:	`std::views::`**`drop_while()`**
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed
Is sized range:	If common random-access range passed
Is common range:	If on common range
Is borrowed range:	If on borrowed view or on lvalue non-view
Caches:	Always caches `begin()`
Const iterable:	Never
Propagates constness:	— (cannot call `begin()` if `const`)

The class template `std::ranges::`**`drop_while_view<>`** defines a view that skips all leading elements of a passed range that match a certain predicate. It yields the opposite elements to the take-while view. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::drop_while_view{rg, [](auto x) {
                                                          return x % 3 != 0;
                                                        }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
3 4 5 6 7 8 9 10 11 12 13
```

Range Adaptors for Drop-While Views

Drop-while views can also (and usually should) be created with a range adaptor. The adaptor simply passes its parameters to the `std::ranges::drop_while_view` constructor:

`std::views::`**`drop_while(`**`*rg*, *pred*`**`)`**

For example:

```
for (const auto& elem : std::views::drop_while(rg, [](auto x) {
                                                    return x % 3 != 0;
                                                  })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::drop_while([](auto x) {
                                                     return x % 3 != 0;
                                                   })) {
  std::cout << elem << ' ';
}
```

The passed predicate must be a callable that satisfies the concept `std::predicate`. This implies the concept `std::regular_invocable`, which means that the predicate should never modify the passed value of the underlying range. However, not modifying the value is a semantic constraint and this cannot always be checked at compile time. As a consequence, the predicate should at least be declared to take the argument by value or by `const` reference.

The drop-while view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The begin iterator is initialized and cached with the first call of `begin()`, which always takes linear time. Therefore, it is better to reuse a drop-while view than creating it again from scratch.

Here is a full example program using drop-while views:

ranges/dropwhileview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                                       //1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::drop_while_view{coll, less4});  //4 1 2 3 4 1
  print(coll | std::views::drop_while(less4));       //4 1 2 3 4 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
4 1 2 3 4 1
4 1 2 3 4 1
```

Drop-While Views and Caching

For better performance (to get an amortized constant complexity), drop-while views cache the result of `begin()` in the view (unless the range is only an input range). This means that the first iteration over the element of a drop-while view is more expensive than further iterations.

For this reason, it is better to initialize a filter view once and use it twice:

```
// better:
auto v1 = coll | std::views::drop_while(myPredicate);
check(v1);
process(v1);
```

than to initialize and use it twice:

```
// worse:
check(coll | std::views::drop_while(myPredicate));
process(coll | std::views::drop_while(myPredicate));
```

The caching may have functional consequences when the range that a filter view refers to is modified. For example:

ranges/dropwhilecache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list   lst{1, 2, 3, 4};

  auto lessThan2 = [](auto v){
    return v < 2;
  };

  auto vVec = vec | std::views::drop_while(lessThan2);
  auto vLst = lst | std::views::drop_while(lessThan2);

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);

  print(vVec);     // OK: 2 3 4
  print(vLst);     // OK: 2 3 4

  // insert more elements at the front (=> 0 98 99 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {0, 98, 99, -1});
  lst.insert(lst.begin(), {0, 98, 99, -1});

  print(vVec);     // OOPS: 99 -1 0 1 2 3 4
  print(vLst);     // OOPS: 2 3 4

  // creating a copy heals (except with random access):
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2);    // OOPS: 99 -1 0 1 2 3 4
  print(vLst2);    // OK: 98 99 -1 0 1 2 3 4
}
```

Note that formally, the copy of the invalid view to the vector creates undefined behavior because the C++ standard does not specify how the caching is done. Because reallocation of a vector invalidates all iterators, a cached iterator would become invalid. However, for random-access ranges, the view usually caches an offset, not the iterator. This means that the view is still valid in that it still contains a valid range although the begin no longer fits the predicate.

As a rule of thumb, **do not use a drop-while view for which `begin()` has been called after the underlying range has been modified**.

Drop-While View and `const`

Note that you ***cannot*** iterate over a `const` drop-while view.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while(...));    // ERROR
```

The problem is that `begin()` is provided only for a non-`const` drop-while view because caching the iterator modifies the view.

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& coll) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while(...));    // OK
```

Interface of Drop-While Views

Table *Operations of the class* `std::ranges::drop_while_view<>` lists the API of a drop-while view.

Operation	Effect
drop_while_view *r*`{}`	Creates a `drop_while_view` that refers to a default constructed range
drop_while_view *r*`{`*rg*`,` *pred*`}`	Creates a `drop_while_view` that refers to all elements of *rg* except the leading elements for which *pred* is `true`
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns
r`.pred()`	Yields a reference to the predicate

Table 8.17. Operations of the class `std::ranges::drop_while_view<>`

8.5.5 Filter View

Type:	`std::ranges::filter_view<>`
Content:	All elements of a range that match a predicate
Adaptor:	`std::views::filter()`
Element type:	Same type as passed range
Requires:	At least input range
Category:	Same as passed but at most bidirectional
Is sized range:	Never
Is common range:	If on common range
Is borrowed range:	Never
Caches:	Always caches `begin()`
Const iterable:	Never
Propagates constness:	— (cannot call `begin()` if `const`)

The class template `std::ranges::filter_view<>` defines a view that iterates only over those elements of the underlying range for which a certain predicate matches. That is, it filters out all elements that do not match the predicate. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::filter_view{rg, [](auto x) {
                                                        return x % 3 != 0;
                                                      }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 2 4 5 7 8 10 11 13
```

Range Adaptors for Filter Views

Filter views can also (and usually should) be created with a range adaptor. The adaptor simply passes its parameters to the `std::ranges::filter_view` constructor:

`std::views::filter(`*`rg`*`, `*`pred`*`)`

For example:

```
for (const auto& elem : std::views::filter(rg, [](auto x) {
                                                  return x % 3 != 0;
                                                })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::filter([](auto x) {
                                                   return x % 3 != 0;
                                                 })) {
  std::cout << elem << ' ';
}
```

The passed predicate must be a callable that satisfies the concept `std::predicate`. This implies the concept `std::regular_invocable`, which means that the predicate should never modify the passed value of the underlying range. However, not modifying the value is a semantic constraint and this cannot always be checked at compile time. As a consequence, the predicate should at least be declared to take the argument by value or by `const` reference.

The filter view is special and you should know when, where, and how to use it, as well as the side effects its use has. In fact, it has a significant impact on the performance of pipelines and restricts write access to elements in a sometimes surprising way.

Therefore, you should use a filter view with care:

- In a pipeline, you should have it as early as possible.
- Be careful with expensive transformations ahead of filters.
- Do not use it for write access to elements when the write access breaks the predicate.

The filter view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The begin iterator is initialized and usually cached with the first call of `begin()`, which always takes linear time. Therefore, it is better to reuse a filter view than to create it again from scratch.

Here is a full example program using filter views:

ranges/filterview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                                 //1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::filter_view{coll, less4}); //1 2 3 1 2 3 1
  print(coll | std::views::filter(less4));      //1 2 3 1 2 3 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
1 2 3 1 2 3 1
1 2 3 1 2 3 1
```

Filter Views and Caching

For better performance (to get an amortized constant complexity), filter views cache the result of `begin()` in the view (unless the range is only an input range). This means that the first iteration over the element of a filter view is more expensive than further iterations.

For this reason, it is better to initialize a filter view once and use it twice:

```
// better:
auto v1 = coll | std::views::filter(myPredicate);
check(v1);
process(v1);
```

than to initialize and use it twice:

```
// worse:
check(coll | std::views::filter(myPredicate));
process(coll | std::views::filter(myPredicate));
```

Note that for ranges that have random access (e.g., arrays, vectors, and deques), the cached offset for the beginning is copied with the view. Otherwise, the cached beginning is not copied.

The caching may have functional consequences when the range that a filter view refers to is modified. For example:

ranges/filtercache.cpp

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list   lst{1, 2, 3, 4};
  auto biggerThan2 = [](auto v){
    return v > 2;
  };

  auto vVec = vec | std::views::filter(biggerThan2);
  auto vLst = lst | std::views::filter(biggerThan2);

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);

  print(vVec);     // OK: 3 4
  print(vLst);     // OK: 3 4

  // insert more elements at the front (=> 98 99 0 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {98, 99, 0, -1});
  lst.insert(lst.begin(), {98, 99, 0, -1});

  print(vVec);     // OOPS: -1 3 4
  print(vLst);     // OOPS: 3 4

  // creating a copy heals (except with random access):
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2);    // OOPS: -1 3 4
  print(vLst2);    // OK: 98 99 3 4
}
```

Note that formally, the copy of the invalid view to the vector creates undefined behavior because the C++ standard does not specify how the caching is done. Because reallocation of a vector invalidates all iterators, a cached iterator would become invalid. However, for random-access ranges, the view usually caches an offset, not the iterator. This means that the view is still valid in that it still contains a valid range although the begin is now the third element regardless of whether it fits the filter.

As a rule of thumb, **do not use a filter view for which `begin()` has been called after the underlying range has been modified**.

Filter Views When Modifying Elements

When using filter views, there is an important additional restriction on write access: you have to ensure that the modified value still fulfills the predicate passed to the filter.[5]

To understand why this is the case, consider the following program:

[5] Thanks to Tim Song for pointing this out.

ranges/viewswrite.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

void print(const auto& coll)
{
  std::cout << "coll: ";
  for (int i : coll) {
    std::cout << i << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<int> coll{1, 4, 7, 10, 13, 16, 19, 22, 25};

  // view for all even elements of coll:
  auto isEven = [] (auto&& i) { return i % 2 == 0; };
  auto collEven = coll | vws::filter(isEven);

  print(coll);

  // modify even elements in coll:
  for (int& i : collEven) {
    std::cout << " increment " << i << '\n';
    i += 1;      // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);

  // modify even elements in coll:
  for (int& i : collEven) {
    std::cout << " increment " << i << '\n';
    i += 1;      // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);
}
```

We iterate twice over the even elements of a collection and increment them. A naive programmer would assume that we get the following output:

```
coll: 1 4 7 10 13 16 19 22 25
coll: 1 5 7 11 13 17 19 23 25
coll: 1 5 7 11 13 17 19 23 25
```

Instead, the program has the following output:

```
coll: 1 4 7 10 13 16 19 22 25
coll: 1 5 7 11 13 17 19 23 25
coll: 1 6 7 11 13 17 19 23 25
```

With the second iteration, we increment the former first even element once again. Why?

The first time we use the view that ensures that we deal only with even elements it works fine. However, the view caches the position of the first element that matches the predicate so that `begin()` does not have to recalculate it. Therefore, when we access the value there, the filter does not apply the predicate again because it already knows that this is the first matching element. Therefore, when we iterate the second time, the filter gives back the former first element. However, for all other elements, we have to perform the check again, which means that the filter finds no more elements because they are all odd now.

There has been some discussion about whether one-pass write access with a filter should be well-formed even if it breaks the predicate of the filter. This is because the requirement that a modification should not violate the predicate of a filter invalidates some very reasonable examples: Here is one of them:[6]

```
for (auto& m : collOfMonsters | filter(isDead)) {
  m.resurrect();      // a shaman's doing, of course
}
```

This code usually compiles and works. However, formally, we have undefined behavior again, because the predicate of the filter ("monsters have to be dead") is broken. Any other "modification" of a (dead) monster would be possible, though (e.g., to "burning" them).

To break the predicate, you have to use an ordinary loop instead:

```
for (auto& m : collOfMonsters) {
  if (m.isDead()) {
    m.resurrect();  // a shaman's doing, of course
  }
}
```

Filter View and `const`

Note that you ***cannot*** iterate over a `const` filter view.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::filter(...));    // ERROR
```

[6] Thanks to Patrice Roy for this example.

The problem is that `begin()` is provided only for a non-`const` filter view because caching the iterator modifies the view.

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& coll) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::filter(...));    // OK
```

Filter Views in Pipelines

When using a filter view in a pipeline, there are a couple of issues to consider:

- In a pipeline, you should have it as early as possible.
- Be careful especially with expensive transformations ahead of filters.
- When using filters while modifying elements, ensure that after these modifications, the predicate of the filter is still satisfied. See below for details.

The reason for this is that views and adaptors in front of the filter view might have to evaluate elements multiple times, once to decide whether they pass the filter and once to finally use their value.

Therefore, a pipeline such as the following:

```
rg | std::views::filter(pred) | std::views::transform(func)
```

has a better performance than

```
rg | std::views::transform(func) | std::views::filter(pred)
```

Interface of Filter Views

Table *Operations of the class* `std::ranges::filter_view<>` lists the API of a filter view.

Filter views never provide `size()`, `data()` or the operator `[]` because they are neither sized nor provide random access.

Operation	**Effect**
filter_view `r{}`	Creates a `filter_view` that refers to a default constructed range
filter_view `r{rg, pred}`	Creates a `filter_view` that refers to all elements or *rg* for which *pred* is `true`
`r.begin()`	Yields the begin iterator
`r.end()`	Yields the sentinel (end iterator)
`r.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (r)`	`true` if *r* is not empty (available if `empty()` is defined)
`r.front()`	Yields the first element (available if forwarding)
`r.back()`	Yields the last element (available if bidirectional and common)
`r.base()`	Yields a reference to the range that *r* refers to or owns
`r.pred()`	Yields a reference to the predicate

Table 8.18. Operations of the class `std::ranges::filter_view<>`

8.6 Transforming Views

This section discusses all views that yield modified values of the elements they iterate over.

8.6.1 Transform View

Type:	`std::ranges::`**`transform_view<>`**
Content:	The transformed values of all element in a range
Adaptor:	`std::views::`**`transform()`**
Element type:	The return type from the transformation
Requires:	At least input range
Category:	Same as passed but at most random access
Is sized range:	If on sized range
Is common range:	If on common range
Is borrowed range:	Never
Caches:	Nothing
Const iterable:	If on const-iterable range and transformation works on const values
Propagates constness:	Only if on rvalue range

The class template `std::ranges::`**`transform_view<>`** defines a view that yields all elements of an underlying range after applying a passed transformation. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
// print all elements squared:
for (const auto& elem : std::ranges::transform_view{rg, [](auto x) {
                                                          return x * x;
                                                        }}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
1 4 9 16 25 36 49 64 81 100 121 144 169
```

Range Adaptors for Transform Views

Transform views can also (and usually should) be created with a range adaptor. The adaptor simply passes its parameters to the `std::ranges::transform_view` constructor.

`std::views::`**`transform(`***`rg`*, *`func`***`)`**

For example:

```
for (const auto& elem : std::views::transform(rg, [](auto x) {
                                                    return x * x;
                                                  })) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::transform([](auto x) {
                                                        x * x;
                                                      })) {
  std::cout << elem << ' ';
}
```

Here is a full example program using transform views:

ranges/transformview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <cmath>    // for std::sqrt()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll);                                    // 1 2 3 4 1
  auto sqrt = [] (auto v) { return std::sqrt(v); };
  print(std::ranges::transform_view{coll, sqrt});  // 1 1.41421 1.73205 2 1
  print(coll | std::views::transform(sqrt));        // 1 1.41421 1.73205 2 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4 1
1 1.41421 1.73205 2 1 1.41421 1.73205 2 1
1 1.41421 1.73205 2 1 1.41421 1.73205 2 1
```

The transformation must be a callable that satisfies the concept `std::regular_invocable`. That implies that transformation should never modify the passed value of the underlying range. However, not modifying the value is a semantic constraint and this cannot always be checked at compile time. As a consequence, the callable should at least be declared to take the argument by value or by `const` reference.

The transform view yields values that have the return type of the transformation. Therefore, in our example, the transform view yields elements of type `double`.

The return type of a transformation might even be a reference. For example:

ranges/transformref.cpp

```cpp
#include <iostream>
#include <vector>
#include <utility>
#include <ranges>

void printPairs(const auto& collOfPairs)
{
  for (const auto& elem : collOfPairs) {
    std::cout << elem.first << '/' << elem.second << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // initialize collection with pairs of int as elements:
  std::vector<std::pair<int,int>> coll{{1,9}, {9,1}, {2,2}, {4,1}, {2,7}};
  printPairs(coll);

  // function that yields the smaller of the two values in a pair:
  auto minMember = [] (std::pair<int,int>& elem) -> int& {
    return elem.second < elem.first ? elem.second : elem.first;
  };

  // increment the smaller of the two values in each pair element:
  for (auto&& member : coll | std::views::transform(minMember)) {
    ++member;
  }
  printPairs(coll);
}
```

The program has the following output:

```
1/9 9/1 2/2 4/1 2/7
2/9 9/2 3/2 4/2 3/7
```

Note that the lambda passed to transform here as the transformation has to specify that it returns a reference:

```cpp
 [] (std::pair<int,int>& elem) -> int& {
     return ...
 }
```

Otherwise, the lambda would return a copy of each element so that the members of these copies are incremented and the elements in `coll` remain unchanged.

The transform view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

Special Characteristics of Transform Views

As usual for standard views, the transform views does not delegate constness to the elements. This means that even when the view is `const` the elements passed to the transforming function are not `const`.

For example, using the example *transformref.cpp* above, you can also implement the following:

```
std::vector<std::pair<int,int>> coll{{1,9}, {9,1}, {2,2}, {4,1}, {2,7}};

// function that yields the smaller of the two values in a pair:
auto minMember = [] (std::pair<int,int>& elem) -> int& {
  return elem.second < elem.first ? elem.second : elem.first;
};

// increment the smaller of the two values in each pair element:
const auto v = coll | std::views::transform(minMember);
for (auto&& member : v) {
  ++member;
}
```

Note that a take view is only common (has same type for iterator and sentinel) if the underlying range is a sized range and a common range. To harmonize types, you might have to use a common view.

Both the begin iterator and the sentinel (end iterator) are special internal helper types.

Interface of Transform Views

Table *Operations of the class* `std::ranges::transform_view<>` lists the API of a transform view.

Operation	**Effect**
transform_view *r*`{}`	Creates a `transform_view` that refers to a default constructed range
transform_view *r*`{`*rg*`,` *func*`}`	Creates a `transform_view` with the values of all elements of range *rg* transformed by *func*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.19. Operations of the class `std::ranges::transform_view<>`

8.6.2 Elements View

Type:	`std::ranges::`**`elements_view<>`**
Content:	The *n*-th member/attribute of all tuple-like elements of a range
Adaptor:	`std::views::`**`elements<>`**
Element type:	Type of the member/attribute
Requires:	At least input range
Category:	Same as passed but at most random access
Is sized range:	If on sized range
Is common range:	If on common range
Is borrowed range:	If on borrowed view or on lvalue non-view
Caches:	Nothing
Const iterable:	If on const-iterable range
Propagates constness:	Only if on rvalue range

The class template `std::ranges::`**`elements_view<>`** defines a view that selects the *idx*-th member/attribute/element of all elements of a passed range. The view calls `get<`*`idx`*`>(`*`elem`*`)` for each element and can especially be used

- To get the *`idx`*-th members of all `std::tuple` elements
- To get the *`idx`*-th alternative of all `std::variant` elements
- To get the `first` or `second` member of `std::pair` elements (although, for elements of associative and unordered containers, it is more convenient to use the `keys_view` and the `values_view`)

Note that for this view, class template argument deduction does not work because you have to specify the index as an argument explicitly and partial argument deduction is not supported for class templates. For this reason, you have to write the following:

```
std::vector<std::tuple<std::string, std::string, int>> rg{
    {"Bach", "Johann Sebastian", 1685}, {"Mozart", "Wolfgang Amadeus", 1756},
    {"Beethoven", "Ludwig van", 1770}, {"Chopin", "Frederic", 1810},
};

for (const auto& elem
      : std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg}) {
    std::cout << elem << ' ';
}
```

The loop prints:

```
1685 1756 1770 1810
```

Note that you have to specify the type of the underlying range with the range adaptor `all()` if a range is passed that is not a view:

```
std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg}      // OK
```

You can also specify the type directly using `std::views::all_t<>`:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, 2>{rg}    // OK
std::ranges::elements_view<std::views::all_t<decltype((rg))>, 2>{rg}  // OK
```

However, details matter here. If the range is not a view yet, the parameter to `all_t<>` must be an lvalue reference. Therefore, you need a `&` after the type of `rg` or the double parentheses around `rg` (by rule, `decltype` yields an lvalue reference if an expression is passed that is an lvalue). Single parentheses without `&` do not work:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)>, 2>{rg}      // ERROR
```

Therefore, the far easier way to declare the view is by using the corresponding range adaptor.

Range Adaptors for Elements Views

Elements views can also (and usually should) be created with a range adaptor. The adaptor makes the use of the view a lot easier because you do not have to specify the type of the underlying range:

```
std::views::elements<idx>(rg)
```

For example:

```
for (const auto& elem : std::views::elements<2>(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::elements<2>) {
  std::cout << elem << ' ';
}
```

Using the range adaptor, `elements<`*idx*`>(`*rg*`)` is always equivalent to:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, idx>{rg}
```

The elements view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

Both the begin iterator and the sentinel (end iterator) are special internal helper types that are common if the passed range is common.

Here is a full example program using element views:

ranges/elementsview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <numbers>     // for math constants
#include <algorithm>   // for sort()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::tuple<int, std::string, double>> coll{
      {1, "pi", std::numbers::pi},
      {2, "e", std::numbers::e},
      {3, "golden-ratio", std::numbers::egamma},
      {4, "euler-constant", std::numbers::phi},
  };

  std::ranges::sort(coll, std::less{},
                    [](const auto& e) {return std::get<2>(e);});
  print(std::ranges::elements_view<decltype(std::views::all(coll)), 1>{coll});
  print(coll | std::views::elements<2>);
}
```

The program has the following output:

```
golden-ratio euler-constant e pi
0.577216 1.61803 2.71828 3.14159
```

Note that you should ***not*** sort the elements of `coll` as follows:

```
std::ranges::sort(coll | std::views::elements<2>);   // OOPS
```

This will sort only the values of the elements, not the elements as whole, and will lead to the following output:

```
pi e golden-ratio euler-constant
0.577216 1.61803 2.71828 3.14159
```

Elements Views for Other Tuple-Like Types

To be able to use this view for user-defined types, you need to specify a tuple-like API for them. However, with the current specification, it is not really due to a problem in the design of tuple-like APIs in general and the way the corresponding concepts are defined. Therefore, strictly speaking, you can use the class `std::ranges::elements_view` or the adaptor `std::views::elements<>` only for `std::pair<>` and `std::tuple<>`.

However, if you ensure that the tuple-like API is defined ***before*** header `<ranges>` is included, it works.

For example:

ranges/elementsviewhack.hpp

```
#include <iostream>
#include <string>
#include <tuple>
// don't include <ranges> yet !!

struct Data {
  int id;
  std::string value;
};

std::ostream& operator<< (std::ostream& strm, const Data& d) {
    return strm << '[' << d.id << ": " << d.value << ']';
}

// tuple-like access to Data:
namespace std {
  template<>
  struct tuple_size<Data> : integral_constant<size_t, 2> {
  };

  template<>
  struct tuple_element<0, Data> {
    using type = int;
  };
  template<>
  struct tuple_element<1, Data> {
    using type = std::string;
  };

  template<size_t Idx> auto get(const Data& d) {
    if constexpr (Idx == 0) {
      return d.id;
    }
    else {
      return d.value;
    }
  }
} // namespace std
```

If you use it as follows:

ranges/elementsviewhack.cpp

```
// don't include <ranges> before "elementsview.hpp"
#include "elementsviewhack.hpp"
#include <iostream>
#include <vector>
#include <ranges>

void print(const auto& coll)
{
  std::cout << "coll:\n";
  for (const auto& elem : coll) {
    std::cout << "- " << elem << '\n';
  }
}

int main()
{
  Data d1{42, "truth"};
  std::vector<Data> coll{d1, Data{0, "null"}, d1};
  print(coll);
  print(coll | std::views::take(2));
  print(coll | std::views::elements<1>);
}
```

you have the following output:

```
coll:
- [42: truth]
- [0: null]
- [42: truth]
coll:
- [42: truth]
- [0: null]
coll:
- truth
- null
- truth
```

Interface of Elements Views

Table *Operations of the class* `std::ranges::elements_view<>` lists the API of an elements view.

Operation	**Effect**
elements_view r{}	Creates an `elements_view` that refers to a default constructed range
elements_view r{*rg*}	Creates an `elements_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.20. Operations of the class `std::ranges::elements_view<>`

8.6.3 Keys and Values View

Type:	`std::ranges::`**`keys_view<>`**	`std::ranges::`**`values_view<>`**
Content:	The first/second member or attribute of all tuple-like elements of a range	
Adaptor:	`std::views::`**`keys`**	`std::views::`**`values`**
Element type:	Type of the first member	Type of the second member
Requires:	At least input range	At least input range
Category:	Same as passed but at most random access	
Is sized range:	If on sized range	If on sized range
Is common range:	If on common range	If on common range
Is borrowed range:	If on borrowed view or on lvalue non-view	
Caches:	Nothing	Nothing
Const iterable:	If on const-iterable range	If on const-iterable range
Propagates constness:	Only if on rvalue range	Only if on rvalue range

The class template `std::ranges::`**`keys_view<>`** defines a view that selects the **first** member/attribute/element from the elements of a passed range. It is nothing but a shortcut for using the elements view with the index `0`. That is, it calls `get<0>(`*elem*`)` for each element.

The class template `std::ranges::`**`values_view<>`** defines a view that selects the **second** member/attribute/element from the elements of a passed range. It is nothing but a shortcut for using the elements view with the index `1`. That is, it calls `get<1>(`*elem*`)` for each element.

These views can especially be used:

- To get the member `first/second` of `std::pair` elements, which is especially useful for selecting the key/value of the elements of a `map`, `unordered_map`, `multimap`, and `unordered_multimap`
- To get the first/second member of `std::tuple` elements
- To get the first/second alternative of `std::variant` elements

However, for the last two applications, it is probably more readable to use the `elements_view` directly with the index `0`.

Note that class template argument deduction does not work yet for these views.[7] For this reason, you have to specify the template parameters explicitly. For example:

```
std::map<std::string, int> rg{
    {"Bach", 1685}, {"Mozart", 1756}, {"Beethoven", 1770},
    {"Tchaikovsky", 1840}, {"Chopin", 1810}, {"Vivaldi", 1678},
};

for (const auto& e : std::ranges::keys_view<decltype(std::views::all(rg))>{rg}) {
    std::cout << e << ' ';
}
```

The loop prints:

```
Bach Beethoven Chopin Mozart Tchaikovsky Vivaldi
```

Range Adaptors for Keys/Values Views

Keys and values views can also (and usually should) be created with a range adaptor: The adaptor makes the use of the view a lot easier because you do not have to specify the type of the underlying range:

```
std::views::keys(rg)
std::views::values(rg)
```

For example:

```
for (const auto& elem : rg | std::views::keys) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::values) {
  std::cout << elem << ' ';
}
```

All other aspects match those of elements views.

[7] See `http://wg21.link/lwg3563`.

Here is a full example program using keys and values views:

ranges/keysvaluesview.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>
#include <numbers>     // for math constants
#include <algorithm>   // for sort()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::unordered_map<std::string, double> coll{
      {"pi", std::numbers::pi},
      {"e", std::numbers::e},
      {"golden-ratio", std::numbers::egamma},
      {"euler-constant", std::numbers::phi},
  };

  print(std::ranges::keys_view<decltype(std::views::all(coll))>{coll});
  print(std::ranges::values_view<decltype(std::views::all(coll))>{coll});

  print(coll | std::views::keys);
  print(coll | std::views::values);
}
```

The program has the following output:

```
euler-constant golden-ratio e pi
1.61803 0.577216 2.71828 3.14159
euler-constant golden-ratio e pi
1.61803 0.577216 2.71828 3.14159
```

8.7 Mutating Views

This section discusses all views that change the order of elements (so far, it is only one view).

8.7.1 Reverse View

Type:	`std::ranges::`**`reverse_view<>`**
Content:	All elements of a range in reverse order
Adaptor:	`std::views::`**`reverse`**
Element type:	Same type as passed range
Requires:	At least bidirectional range
Category:	Same as passed but at most random access
Is sized range:	If on sized range
Is common range:	Always
Is borrowed range:	If on borrowed view or on lvalue non-view
Caches:	Caches `begin()` unless common range or random access and sized range
Const iterable:	If on const-iterable common range
Propagates constness:	Only if on rvalue range

The class template `std::ranges::`**`reverse_view<>`** defines a view that iterates over the elements of the underlying range in the opposite order.

For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::reverse_view{rg}) {
  std::cout << elem << ' ';
}
```

The loop prints:

```
13 12 11 10 9 8 7 6 5 4 3 2 1
```

Range Adaptors for Reverse Views

Reverse views can also (and usually should) be created with a range adaptor:

`std::views::`**`reverse(`***`rg`***`)`**

For example:

```
for (const auto& elem : std::views::reverse(rg)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : rg | std::views::reverse) {
  std::cout << elem << ' ';
}
```

Note that the adaptor might not always yield a `reverse_view`:

- A reversed reversed range yields the original range.
- If a reversed subrange is passed, the original subrange is returned with the corresponding non-reverse iterators.

The reverse view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

The iterators are just reverse iterators of the passed range.

Here is a full example program using reverse views:

ranges/reverseview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4};

  print(coll);                              //1 2 3 4 1 2 3 4
  print(std::ranges::reverse_view{coll});   //4 3 2 1 4 3 2 1
  print(coll | std::views::reverse);        //4 3 2 1 4 3 2 1
}
```

The program has the following output:

```
1 2 3 4 1 2 3 4
4 3 2 1 4 3 2 1
4 3 2 1 4 3 2 1
```

Reverse Views and Caching

For better performance, reverse views cache the result of `begin()` in the view (unless the range is only an input range).

Note that for ranges that have random access (e.g., arrays, vectors, and deques), the cached offset for the beginning is copied with the view. Otherwise, the cached beginning is not copied.

The caching may have functional consequences when the range that a reverse view refers to is modified. For example:

ranges/reversecache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list   lst{1, 2, 3, 4};

  auto vVec = vec | std::views::take(3) | std::views::reverse;
  auto vLst = lst | std::views::take(3) | std::views::reverse;
  print(vVec);                  // OK: 3 2 1
  print(vLst);                  // OK: 3 2 1

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);
  print(vVec);                  // OK: 2 1 0
  print(vLst);                  // OOPS: 3 2 1

  // creating a copy heals:
  auto vVec2 = vVec;
  auto vLst2 = vLst;
  print(vVec2);                 // OK: 2 1 0
  print(vLst2);                 // OK: 2 1 0
}
```

Note that formally, the copy of the view to the vector creates undefined behavior because the C++ standard does not specify how the caching is done. Because reallocation of a vector invalidates all iterators, a cached iterator would become invalid. However, for random-access ranges, the view usually caches an offset, not the iterator. Therefore, the view is still valid for the vector.

Note also that a reverse view on a container as a whole works fine because the end iterator is cached.

As a rule of thumb, **do not use a reverse view for which `begin()` has been called after the underlying range has been modified**.

Reverse View and `const`

Note that you ***cannot always*** iterate over a `const` reverse view. In fact, the referenced range has to be a common range.

For example:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
                                                  return x % 2 != 0;
                                                });

printElems(vec | std::views::reverse);           // OK
printElems(vecFirstOdd);                         // OK
printElems(vecFirstOdd | std::views::reverse);   // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& coll) {
  ...
}

std::vector vec{1, 2, 3, 4, 5};

// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
                                                  return x % 2 != 0;
                                                });

printElems(vecFirstOdd | std::views::reverse);   // OK
```

Interface of Reverse Views

Table *Operations of the class* `std::ranges::reverse_view<>` lists the API of a reverse view.

Operation	Effect
reverse_view *r*`{}`	Creates a `reverse_view` that refers to a default constructed range
reverse_view *r*`{`*rg*`}`	Creates a `reverse_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.21. Operations of the class `std::ranges::reverse_view<>`

8.8 Views for Multiple Ranges

This section discusses all views that deal with multiple ranges.

8.8.1 Split and Lazy-Split View

Type:	`std::ranges::`**`split_view<>`**	`std::ranges::`**`lazy_split_view<>`**
Content:	All elements of a range split into multiple views	
Adaptor:	`std::views::`**`split()`**	`std::views::`**`lazy_split()`**
Element type:	Collection of references	Collection of references
Requires:	At least forward range	At least input range
Category:	Always forward	Input or forward
Is sized range:	Never	Never
Is common range:	If on common forward range	If on common forward range
Is borrowed range:	Never	Never
Caches:	Always caches `begin()`	Caches the current value if on input range
Const iterable:	Never	If on const-iterable forward range
Propagates constness:	—	Never

Both class templates `std::ranges::`**`split_view<>`** and `std::ranges::`**`lazy_split_view<>`** define a view that refers to multiple sub-views of a range separated by a passed separator.[8]

The difference between `split_view<>` and `lazy_split_view<>` is as follows:

- `split_view<>` cannot iterate over a `const` view; `lazy_split_view<>` can (if it refers to a range that is at least a forward range).
- `split_view<>` can only deal with ranges that have at least forward iterators (concept `forward_range` has to be satisfied).
- `split_view<>` elements are just a `std::ranges::subrange` of the iterator type of the referred range (keeping category of the referred range).
  `lazy_split_views<>` elements are views of type `std::ranges::lazy_split_view` that are always forward ranges, meaning that even `size()` is not supported.
- `split_view<>` has the better performance.

This means that you should usually use `split_view<>` unless you cannot do that due to using an input range only or the view is used as a `const` range. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

[8] `std::ranges::lazy_split_view<>` was not part of the original C++20 standard but was added to C++20 afterwards with `http://wg21.link/p2210r2`.

The loops print:

```
1 2 3 4
6 7 8 9 10 11 12 13
```

That is, wherever we find an element with value 5 in rg, we end the previous view and start a new view.

Range Adaptors for Split and Lazy-Split Views

Split and lazy-split views can (and usually should) also be created with range adaptors. The adaptors simply pass their parameters to the corresponding view constructor:

std::views::**split**(*rg*, *sep*)

std::views::**lazy_split**(*rg*, *sep*)

For example:

```
for (const auto& sub : std::views::split(rg, 5)) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

or:

```
for (const auto& sub : rg | std::views::split(5)) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

The views created might be empty. Thus, for each leading and trailing separator, and whenever two separators are behind each other, an empty view is created. For example:

```
std::list<int> rg{5, 5, 1, 2, 3, 4, 5, 6, 5, 5, 4, 3, 2, 1, 5, 5};
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  std::cout << "subview: ";
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Here, the output is as follows:[9]

```
subview:
subview:
```

[9] The original C++20 standard specified that the last separator is ignored, meaning that we would get only one empty sub-view at the end. This was fixed with http://wg21.link/p2210r2.

```
subview: 1 2 3 4
subview: 6
subview:
subview: 4 3 2 1
subview:
subview:
```

Instead of a single value, you can also pass a sequence of values that serves as a separator. For example:

```
std::vector<int> rg{1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3 };
...
// split separated by a sequence of 5 and 1:
for (const auto& sub : std::views::split(rg, std::list{5, 1})) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

The output of this code is

```
1 2 3 4
2 3 4
2 3
```

The passed collection of elements must be a valid view and a `forward_range`. Therefore, when specifying the sub-sequence in a container, you have to convert it into a view. For example:

```
// split with specified pattern of 4 and 5:
std::array pattern{4, 5};
for (const auto& sub : std::views::split(rg, std::views::all(pattern))) {
  ...
}
```

You can use split views to split strings. For example:

```
std::string str{"No problem can withstand the assault of sustained thinking"};
for (auto sub : std::views::split(str, "th"sv)) {  // split by "th"
  std::cout << std::string_view{sub} << '\n';
}
```

Each sub-string `sub` is of type `std::ranges::subrange<decltype(str.begin())>`. Code like this will not work with a lazy-split view.

A split or lazy-split view stores the passed range internally (optionally converted to a view in the same way as with `all()`). Therefore, it is valid only as long as the passed range is valid (unless an rvalue was passed, meaning that internally, an owning view is used).

Both the begin iterator and the sentinel (end iterator) are special internal helper types that are common if the passed range is common.

Here is a full example program using split views:

ranges/splitview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <array>
#include <ranges>

void print(auto&& obj, int level = 0)
{
  if constexpr(std::ranges::input_range<decltype(obj)>) {
    std::cout << '[';
    for (const auto& elem : obj) {
       print(elem, level+1);
    }
    std::cout << ']';
  }
  else {
    std::cout << obj << ' ';
  }
  if (level == 0) std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4};

  print(coll);                              // [1 2 3 4 1 2 3 4 ]
  print(std::ranges::split_view{coll, 2}); // [[1 ][3 4 1 ][3 4 ]]
  print(coll | std::views::split(3));       // [[1 2 ][4 1 2 ][4 ]]
  print(coll | std::views::split(std::array{4, 1})); // [[1 2 3 ][2 3 4 ]]
}
```

The program has the following output:

```
[1 2 3 4 1 2 3 4 ]
[[1 ][3 4 1 ][3 4 ]]
[[1 2 ][4 1 2 ][4 ]]
[[1 2 3 ][2 3 4 ]]
```

Split View and `const`

Note that you ***cannot*** iterate over a `const` split view.

For example:

```
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::split_view sv{coll, 5};
for (const auto& sub : sv) {                    // ERROR for const split view
  std::cout << sub.size() << ' ';
}
```

This is a problem especially if you pass the view to a generic function that takes the parameter as a `const` reference:

```
void printElems(const auto& coll) {
  ...
}

printElems(std::views::split(rg, 5));           // ERROR
```

To support this view in generic code, you have to use universal/forwarding references:

```
void printElems(auto&& coll) {
  ...
}
```

Alternatively, you can use a `lazy_split_view`. Then, however, the sub-view elements can only be used as forward ranges, meaning that you cannot do things like calling `size()`, iterating backward, or sorting the elements:

```
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::lazy_split_view sv{coll, 5};
for (const auto& sub : sv) {                    // OK for const lazy-split view
  std::cout << sub.size() << ' ';               // ERROR
  std::sort(sub);                               // ERROR
  for (const auto& elem : sub) {                // OK
    std::cout << elem << ' ';
  }
}
```

Interface of Split and Lazy-Split Views

Table *Operations of the classes* `std::ranges::split_view<>` *and* `std::ranges::split_view<>` lists the API of a split or lazy-split view.

Operation	**Effect**
split_view *r*{}	Creates a `split_view` that refers to a default constructed range
split_view *r*{*rg*}	Creates a `split_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.22. Operations of the classes `std::ranges::split_view<>` *and* `std::ranges::split_view<>`

8.8.2 Join View

Type:	`std::ranges::join_view<>`
Content:	All elements of a range of multiple ranges as one view
Adaptor:	`std::views::join()`
Element type:	Same type as passed ranges
Requires:	At least input range
Category:	Input to bidirectional
Is sized range:	Never
Is common range:	Varies
Is borrowed range:	Never
Caches:	Nothing
Const iterable:	If on const-iterable range and elements are still references with `const`
Propagates constness:	Only if on rvalue range

The class template `std::ranges::join_view<>` defines a view that iterates over all elements of a view of multiple ranges.

For example:

```
std::vector<int> rg1{1, 2, 3, 4};
std::vector<int> rg2{0, 8, 15};
std::vector<int> rg3{5, 4, 3, 2, 1, 0};
std::array coll{rg1, rg2, rg3};
...
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

The loops print:

```
1 2 3 4 0 8 15 5 4 3 2 1 0
```

Range Adaptors for Join Views

Join views can also (and usually should) be created with a range adaptor. The adaptor simply passes its parameters to the `std::ranges::join_view` constructor:

`std::views::join(`*`rg`*`)`

For example:

```
for (const auto& elem : std::views::join(coll)) {
  std::cout << elem << ' ';
}
```

or:

```
for (const auto& elem : coll | std::views::join) {
  std::cout << elem << ' ';
}
```

Here is a full example program using join views:

ranges/joinview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <array>
#include <ranges>
#include "printcoll.hpp"

int main()
{
  std::vector<std::string> rg1{"he", "hi", "ho"};
  std::vector<std::string> rg2{"---", "|", "---"};
  std::array coll{rg1, rg2, rg1};

  printColl(coll);                                       // ranges of ranges of strings
  printColl(std::ranges::join_view{coll});               // range of strings
  printColl(coll | std::views::join);                    // range of strings
  printColl(coll | std::views::join | std::views::join); // ranges of chars
}
```

It uses a helper function that can print collections recursively:

ranges/printcoll.hpp

```cpp
#include <iostream>
#include <ranges>

template<typename T>
void printColl(T&& obj, int level = 0)
{
  if constexpr(std::same_as<std::remove_cvref_t<T>, std::string>) {
    std::cout << "\"" << obj << "\"";
  }
  else if constexpr(std::ranges::input_range<T>) {
    std::cout << '[';
    for (auto pos = obj.begin(); pos != obj.end(); ++pos) {
       printColl(*pos, level+1);
       if (std::ranges::next(pos) != obj.end()) {
         std::cout << ' ';
       }
    }
    std::cout << ']';
  }
```

```
    else {
      std::cout << obj;
    }
    if (level == 0) std::cout << '\n';
  }
```

The program has the following output:

```
[["he" "hi" "ho"] ["---" "|" "---"] ["he" "hi" "ho"]]
["he" "hi" "ho" "---" "|" "---" "he" "hi" "ho"]
["he" "hi" "ho" "---" "|" "---" "he" "hi" "ho"]
[h e h i h o - - - | - - - h e h i h o]
```

Together with type `std::ranges::subrange`, you can use a join view to join elements of multiple arrays. For example:

```
int arr1[]{1, 2, 3, 4, 5};
int arr2[] = {0, 8, 15};
int arr3[10]{1, 2, 3, 4, 5};
...
std::array<std::ranges::subrange<int*>, 3> coll{arr1, arr2, arr3};
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

Alternatively, you can declare `coll` as follows:

```
std::array coll{std::ranges::subrange{arr1},
                std::ranges::subrange{arr2},
                std::ranges::subrange{arr3}};
```

The join view is the only view in C++20 that deals with ranges of ranges. Therefore, it has outer and inner iterators. And the properties of the resulting view might depend on both.

Note that the internal iterators do not support iterator traits. For this reason, you should prefer the utilities like `std::ranges::next()` over `std::next()`. Otherwise, code might not compile.

Join View and `const`

Note that you **cannot always** iterate over a `const` join view. This happens if the ranges are not `const` iterable or if the inner ranges yield plain values instead of references.

For the latter case, consider the following example:

ranges/joinconst.cpp

```
#include <vector>
#include <array>
#include <ranges>
#include "printcoll.hpp"

void printConstColl(const auto& coll)
```

```
{
  printColl(coll);
}

int main()
{
  std::vector<int> rg1{1, 2, 3, 4};
  std::vector<int> rg2{0, 8, 15};
  std::vector<int> rg3{5, 4, 3, 2, 1, 0};
  std::array coll{rg1, rg2, rg3};

  printConstColl(coll);
  printConstColl(coll | std::views::join);

  auto collTx = [] (const auto& coll) { return coll; };
  auto coll2values = coll | std::views::transform(collTx);

  printConstColl(coll2values);
  printConstColl(coll2values | std::views::join);    // ERROR
}
```

When we use join the elements of the array of three ranges, we can call `printConstColl()`, which takes the range as a `const` reference. We get the following output:

```
[[1 2 3 4] [0 8 15] [5 4 3 2 1 0]]
[1 2 3 4 0 8 15 5 4 3 2 1 0]
```

However, when we create we pass the whole array to a transform view that yields all inner ranges back by value, calling `printConstColl()` is an error.

Calling `printColl()` for the view, for which the inner ranges yield plain values, works fine. Note that this requires that `printColl()` uses `std::ranges::next()` instead of `std::next()`. Otherwise, even the following doe not compile:

```
printColl(coll2values | std::views::join);    // ERROR if std::next() used
```

Special Characteristics of Join Views

The resulting category is bidirectional if both the outer and the inner ranges are at least bidirectional and the inner ranges are common ranges. Otherwise, the resulting category is forward if both the outer and the inner ranges are at least forward ranges. Otherwise, the resulting category is input range.

Interface of Join Views

Table *Operations of the class* `std::ranges::join_view<>` lists the API of a join view.

Operation	**Effect**
join_view *r*{}	Creates a `join_view` that refers to a default constructed range
join_view *r*{*rg*}	Creates a `join_view` that refers to range *rg*
r`.begin()`	Yields the begin iterator
r`.end()`	Yields the sentinel (end iterator)
r`.empty()`	Yields whether *r* is empty (available if the range supports it)
`if (`*r*`)`	`true` if *r* is not empty (available if `empty()` is defined)
r`.size()`	Yields the number of elements (available if it refers to a sized range)
r`.front()`	Yields the first element (available if forwarding)
r`.back()`	Yields the last element (available if bidirectional and common)
r`[`*idx*`]`	Yields the *n*-th element (available if random access)
r`.data()`	Yields a raw pointer to the memory of the elements (available if elements are in contiguous memory)
r`.base()`	Yields a reference to the range that *r* refers to or owns

Table 8.23. Operations of the class `std::ranges::join_view<>`

Chapter 9

Spans

To deal with ranges, C++20 introduced a couple of view types. Usually, these view types do not store elements in their own memory; they refer to elements stored in other ranges or views. The `std::span<>` class template is one of these views.

Historically, however, spans are a generalization of string views, which were introduced in C++17. Spans refer to arbitrary arrays of any element type. Being just a combination of a raw pointer and a size, they provide the usual interface of collections for reading and writing elements that are stored in contiguous memory.

By requiring that spans can refer only to elements in contiguous memory, the iterators can just be raw pointers, which makes them cheap. It also means that the collection provides random access (so that you can jump to any position of the range), which means that you can use this *view* to sort elements or you can use operations that yield a subsequence of `n` elements located in the middle or at the end of the underlying range.

Using a span is cheap and fast (you should always pass it by value). However, it is also potentially dangerous because, in the same way as for raw pointers, it is up to the programmer to ensure that the referred element sequence is still valid when using a span. In addition, the fact that a span supports write access can introduce situations where `const` correctness is broken (or at least does not work in the way you might expect).

9.1 Using Spans

Let us look at some first examples of using spans. First, however, we have to discuss how the number of elements in a span can be specified.

9.1.1 Fixed and Dynamic Extent

When you declare a span, you can choose between a specified fixed number of elements or leave the number of elements open so that the number of elements the span refers to can change.

A span with a specified fix number of elements is called a span with *fixed extent*. It is declared by specifying the element type and the size as template parameters or by initializing it either with an array (raw array or `std::array<>`) with an iterator and a size:

```
int a5[5] = {1, 2, 3, 4, 5};
std::array arr5{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span sp1 = a5;                    // span with fixed extent of 5 elements
std::span sp2{arr5};                   // span with fixed extent of 5 elements
std::span<int, 5> sp3 = arr5;          // span with fixed extent of 5 elements
std::span<int, 3> sp4{vec};            // span with fixed extent of 3 elements
std::span<int, 4> sp5{vec.data(), 4};  // span with fixed extent of 4 elements
std::span sp6 = sp1;                   // span with fixed extent of 5 elements
```

For such a span, the member function `size()` always yields the size specified as part of the type. The default constructor cannot be called here (unless the extent is `0`).

A span where the number of elements is not stable over its lifetime is called a span with *dynamic extent*. The number of elements depends on the sequence of elements the span refers to and might change due to the assignment of a new underlying range (there is no other way to change the number of elements). For example:

```
int a5[5] = {1, 2, 3, 4, 5};
std::array arr5{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span<int> sp1;                    // span with dynamic extent (initially 0 elements)
std::span sp2{a5, 3};                  // span with dynamic extent (initially 3 elements)
std::span<int> sp3{arr5};              // span with dynamic extent (initially 5 elements)
std::span sp4{vec};                    // span with dynamic extent (initially 8 elements)
std::span sp5{arr5.data(), 3};         // span with dynamic extent (initially 3 elements)
std::span sp6{a5+1, 3};                // span with dynamic extent (initially 3 elements)
```

Note that it is up to the programmer to ensure that the span refers to a valid range that has enough elements.

For both cases, let us look at complete examples.

9.1.2 Example Using a Span with a Dynamic Extent

Here is a first example using a span with dynamic extent:

lib/spandyn.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>
```

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

  // define view to first 3 elements:
  std::span<const std::string> sp{vec.data(), 3};
  std::cout << "first 3:                   ";
  printSpan(sp);

  // sort elements in the referenced vector:
  std::ranges::sort(vec);
  std::cout << "first 3 after sort():      ";
  printSpan(sp);

  // insert a new element:
  // - must reassign the internal array of the vector if it reallocated new memory
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp = std::span{vec.data(), 3};
  }
  std::cout << "first 3 after push_back(): ";
  printSpan(sp);

  // let span refer to the vector as a whole:
  sp = vec;
  std::cout << "all:     ";
  printSpan(sp);

  // let span refer to the last five elements:
  sp = std::span{vec.end()-5, vec.end()};
  std::cout << "last 5: ";
  printSpan(sp);

  // let span refer to the last four elements:
```

```
  sp = std::span{vec}.last(4);
  std::cout << "last 4: ";
  printSpan(sp);
}
```

The output of the program is as follows:

```
first 3:                   "New York" "Tokyo" "Rio"
first 3 after sort():      "Berlin" "New York" "Rio"
first 3 after push_back(): "Berlin" "New York" "Rio"
all:    "Berlin" "New York" "Rio" "Sydney" "Tokyo" "Cairo"
last 5: "New York" "Rio" "Sydney" "Tokyo" "Cairo"
last 4: "Rio" "Sydney" "Tokyo" "Cairo"
```

Let us go through the example step by step.

Declaring a Span

In `main()`, we first initialize a span of three constant strings with the first three elements of a vector:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

std::span<const std::string> sp{vec.data(), 3};
```

For initialization, we pass the beginning of the sequence and the number of elements. In this case, we refer to the first three elements of `vec`.

There are a lot of things to note about this declaration:

- It is up to the programmer to ensure that the number of elements matches the extent of the span and that the elements are valid. If the vector does not have enough elements, the behavior is undefined:

  ```
  std::span<const std::string> sp{vec.begin(), 10};    // undefined behavior
  ```

- By specifying that the elements are `const std::string`, we cannot modify them via the span. Note that declaring the span itself as `const` does ***not*** provide read-only access to the referenced elements (as usual for views, const is not propagated):

  ```
  std::span<const std::string> sp1{vec.begin(), 3};    // elements cannot be modified
  const std::span<std::string> sp2{vec.begin(), 3};    // elements can be modified
  ```

- Using a different element type for the span than for the referenced elements looks makes it look like you can use any type for a span that converts to the underlying element type. However, that is not true. You can only add qualifiers such as `const`:

  ```
  std::vector<int> vec{...};
  std::span<long> sp{vec.data(), 3};                   // ERROR
  ```

Passing and Printing a Span

Next, we print the span, passing it to a generic print function:

```
printSpan(sp);
```

The print function can deal with any span (as long as an output operator is defined for the elements):

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}
```

You might be surprised that the function template `printSpan<>()` can be called even though it has a non-type template parameter for the size of the span. It works because a `std::span<T>` is a shortcut for a span that has the pseudo size `std::dynamic_extent`:

```
std::span<int> sp;   // equivalent to std::span<int, std::dynamic_extent>
```

In fact, the class template `std::span<>` is declared as follows:

```
namespace std {
  template<typename ElementType, size_t Extent = dynamic_extent>
  class span {
    ...
  };
}
```

This allows programmers to provide generic code like `printSpan<>()` that works for both spans with fixed extent and spans with dynamic extent. When calling `printSpan<>()` with a span with a fixed extent, the extent is passed as a template parameter:

```
std::span<int, 5> sp{...};

printSpan(sp);        // calls printSpan<int, 5>(sp)
```

As you can see, the span is passed by value. That is the recommended way to pass spans because they are cheap to copy because internally, a span is just a pointer and a size.

Inside the print function, we use a range-based `for` loop to iterate over the elements of the span. This is possible because a span provides iterator support with `begin()` and `end()`.

However, beware: regardless of we pass the span by value or by `const` reference, inside the function we can still modify the elements as long as they are not declared to be `const`. That is why declaring the elements of the span to be `const` usually makes sense.

Dealing with Reference Semantics

Next, we sort the elements that the span refers to (we use the new `std::ranges::sort()` here, which takes the container as a whole):

```
std::ranges::sort(vec);
```

Because the span has reference semantics, this sorting also affects the elements of the span, with the result that the span now refers to different values.

If we did not have a span of `const` elements, we would also be able to call `sort()` passing the span.

Reference semantics means that you have to be careful when using a span, which demonstrates the next statements in the example. Here, we insert a new element into the vector that holds the elements the span refers to. Due to the reference semantics of a span, this is something we have to be very careful about, because if the vector allocates new memory, it invalidates all iterators and pointers to its elements. Therefore, a reallocation also invalidates a span that refers to the elements of the vector. The span refers to elements that are no longer there.

For this reason, we double check the capacity (maximum number of elements for which memory is allocated) before and after the insertion. If the capacity changes, we reinitialize the span to refer to the first three elements in the new memory:

```
auto oldCapa = vec.capacity();
vec.push_back("Cairo");
if (oldCapa != vec.capacity()) {
  sp = std::span{vec.data(), 3};
}
```

We can only perform this reinitialization because the span itself is not `const`.

Assigning a Container to a Span

Next, we assign the vector as a whole to the span and print it out:

```
std::span<const std::string> sp{vec.begin(), 3};
...
sp = vec;
```

Here, you can see that an assignment to the span with dynamic extent can change the number of elements. Spans can take any type of container or range that hold the elements in contiguous memory, provided the container yields access to the elements with the member function `data()`.

However, due to the limitations of template type deduction, you cannot pass such a container to a function that expects a span. You have to specify explicitly that you want to convert a vector to a span:

```
printSpan(vec);              // ERROR: template type deduction doesn't work here
printSpan(std::span{vec});   // OK
```

Assigning a Different Subsequence

In general, the assignment operator of a span allows us to assign another sequence of elements.

The example uses this to later refer to the last three elements in the vector:

```
std::span<const std::string> sp{vec.data(), 3};
...
// assign view to last five elements:
sp = std::span{vec.end()-5, vec.end()};
```

Here, you can also see that we can specify the referenced sequence with two iterators defining the beginning and the end of the sequence as a half-open range (the value at the beginning included, the value at the end excluded). The requirement is that the concept `std::sized_sentinel_for` is satisfied for the beginning and end so that the constructor can compute the difference.

However, as the following statement demonstrates, the last *n* elements can also be assigned using the member functions of spans:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span<const std::string> sp{vec.data(), 3};
...
// assign view to last four elements:
sp = std::span{vec}.last(4);
```

Spans are the only views that provide a way to yield a sequence of elements in the middle or at the end of a range.

As long as the element type fits, you can pass any sequence of elements of any other type. For example:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span<const std::string> sp{vec.begin(), 3};
...
std::array<std::string, 3> arr{"Tick", "Trick", "Track"};
sp = arr;   // OK
```

However, note that spans do not support implicit type conversions for the element types (except adding `const`). For example, the following code does not compile:

```
std::span<const std::string> sp{vec.begin(), 3};
...
std::array arr{"Tick", "Trick", "Track"};   // deduces std::array<const char*, 3>
sp = arr;                                   // ERROR: different element types
```

9.1.3 Example Using a Span with Non-`const` Elements

When initializing spans, we can use class template argument deduction so that the type of the elements (and the extent) is deduced:

```
std::span sp{vec.begin(), 3};     // deduces: std::span<std::string>
```

The span then declares the type of the elements to have the element type of the underlying range, which means that you can even modify the value of the underlying range provided the underlying range does not declare its elements as `const`.

This feature can be used to allow spans to modify elements of a range in one statement. For example, you can sort a subset of the elements, as the following example demonstrates:

lib/spanview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
  print(vec);

  // sort the three elements in the middle:
  std::ranges::sort(std::span{vec}.subspan(1, 3));
  print(vec);

  // print last three elements:
  print(std::span{vec}.last(3));
}
```

Here, we create temporary spans to sort a subset of the elements in the vector `vec` and print the last three elements of the vector.

The program has the following output:

```
"New York" "Tokyo" "Rio" "Berlin" "Sydney"
"New York" "Berlin" "Rio" "Tokyo" "Sydney"
"Rio" "Tokyo" "Sydney"
```

Spans are views. To deal with the first *n* elements of a range, you can also use the range factory `std::views::counted()`, which creates a span with dynamic extent if called for an iterator to a range with elements in contiguous memory:

```cpp
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::views::counted(vec.begin()+1, 3);   // span with 2nd to 4th elem of vec
```

9.1.4 Example Using a Span with Fixed Extent

As a first example of a span with a fixed extent, let us convert the previous example but declare the span with a fixed extent:

lib/spanfix.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

  // define view to first 3 elements:
  std::span<const std::string, 3> sp3{vec.data(), 3};
  std::cout << "first 3:              ";
  printSpan(sp3);

  // sort referenced elements:
  std::ranges::sort(vec);
  std::cout << "first 3 after sort(): ";
  printSpan(sp3);

  // insert a new element:
  // - must reassign the internal array of the vector if it reallocated new memory
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp3 = std::span<std::string, 3>{vec.data(), 3};
  }
  std::cout << "first 3: ";
  printSpan(sp3);
  // let span refer to the last three elements:
  sp3 = std::span<const std::string, 3>{vec.end()-3, vec.end()};
  std::cout << "last 3:  ";
  printSpan(sp3);

  // let span refer to the last three elements:
  sp3 = std::span{vec}.last<3>();
  std::cout << "last 3:  ";
  printSpan(sp3);
}
```

The output of the program is as follows:

```
first 3:                "New York" "Tokyo" "Rio"
first 3 after sort(): "Berlin" "New York" "Rio"
first 3: "Berlin" "New York" "Rio"
last 3:  "Sydney" "Tokyo" "Cairo"
last 3:  "Sydney" "Tokyo" "Cairo"
```

Again, let us go through the remarkable parts of the program example step by step.

Declaring a Span

This time, we first initialize a span of three constant strings with fixed extent:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

std::span<const std::string, 3> sp3{vec.data(), 3};
```

For the fixed extent, we specify both the type of the elements and the size.

Again, it is up to the programmer to ensure that the number of elements matches the extent of the span. If the count passed as the second argument does not match the extent, the behavior is undefined.

```
std::span<const std::string, 3> sp3{vec.begin(), 4};   // undefined behavior
```

Assigning a Different Subsequence

For spans with fixed extent, you can only assign new underlying ranges with the same number of elements. Therefore, this time, we assign only spans with three elements:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span<const std::string, 3>{vec.end()-3, vec.end()};
```

Note that the following would no longer compile:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span{vec}.last(3);       // ERROR
```

The reason for this is that the expression on the right hand side of the assignment creates a span with dynamic extent. However, by using `last()` with a syntax that specifies the number of elements as template parameters, we get a span with the corresponding fixed extent:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span{vec}.last<3>();     // OK
```

We can still use class template argument deduction to assign the elements of an array or even assign it directly:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
std::array<std::string, 3> arr{"Tic", "Tac", "Toe"};
sp3 = std::span{arr};               // OK
sp3 = arr;                          // OK
```

9.1.5 Fixed vs. Dynamic Extent

Both fixed and dynamic extent have their benefits.

Specifying a fixed size enables compilers to detect size violations at runtime or even at compile time. For example, you cannot assign a `std::array<>` with the wrong number of elements to a span with a fixed extent:

```
std::vector vec{1, 2, 3};
std::array arr{1, 2, 3, 4, 5, 6};

std::span<int, 3> sp3{vec};
std::span sp{vec};

sp3 = arr;                  // compile-time ERROR
sp = arr;                   // OK
```

Spans with a fixed extent also need less memory because they do not need to have a member for the actual size (the size is part of the type).

Using spans with dynamic extent provides more flexibility:

```
std::span<int> sp;          // OK
...
std::vector vec{1, 2, 3, 4, 5};
sp = vec;                   // OK (span has 5 elements)
sp = {vec.data()+1, 3};     // OK (span has 3 elements)
```

9.2 Spans Considered Harmful

Spans refer to a sequence of external values. They therefore have the usual problems that types with reference semantics have. It is up to the programmer to ensure that the sequence a span refers to is valid.

Errors happen pretty fast. For example, if a function `getData()` returns a collection of `int`s by value (e.g., as vector, `std::array`, or raw array), the following statements create fatal runtime errors:

```
std::span<int, 3> first3{getData()};      // ERROR: reference to temporary object

std::span sp{getData().begin(), 3};       // ERROR: reference to temporary object

sp = getData();                           // ERROR: reference to temporary object
```

This can look a little bit more subtle, such as using the range-based `for` loop:

```
// for the last 3 returned elements:
for (auto s : std::span{arrayOfConst()}.last(3))     // fatal runtime ERROR
```

This code causes undefined behavior because due to a bug in the range-based `for` loop, iterating over a reference to a temporary object uses the values when they are already destroyed (see `http://wg21.link/p2012` for details).

Compilers could detect these problems with special "lifetime checks" for standard types, which is currently being implemented for major compilers. However, that can detect only simple lifetime dependencies like the one between a span and the object it is initialized with.

In addition, you have to ensure that the referenced sequence of elements remains valid. This can be a problem if other parts of the program end the lifetime of a referred sequence while the span is still in use.

If we refer into objects (such as into a vector), this validity issue can even happen while the vector is still alive. Consider:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span sp{vec};                     // view to the memory of vec
...
vec.push_back(9);                      // might reallocate memory
std::cout << sp[0];                    // fatal runtime ERROR (referenced array no longer valid)
```

As a workaround, you have to reinitialize the span with the original vector.

In general, using spans is as dangerous as using raw pointers and other view types. Use them with care.

9.3 Design Aspects of Spans

Designing a type that refers to a sequence of elements is not easy. A lot of aspects and trade-offs have to be thought out and decided on:

- Performance versus safety
- `const` correctness
- Possible implicit and explicit type conversions
- Requirements for the supported types
- Supported APIs

Let me first clarify what a span not is:

- A span is not a container. It might have several aspects of a container (e.g., the ability to iterate over elements with `begin()` and `end()`), but there are already several issues due to its reference semantics:
  – Should elements also be `const` if the span is `const`?
  – What does an assignment do: assign a new sequence or assign new values to the referenced elements?
  – Should we provide `swap()` and what does it do?
- A span is not a pointer (with a size). It does not make sense to provide the operators `*` and `->`.

Type `std::span` is a very specific reference to a sequence of elements. And it is important to understand these specific aspects to use this type correctly.

There is a very helpful blog post by Barry Revzin about this that I strongly recommend you to read: `http://brevzin.github.io/c++/2018/12/03/span-best-span/`.

Note that C++20 provides also other ways for dealing with references to (sub-)sequences, such as subranges. They also work for sequences that are not stored in contiguous memory. By using the range factory `std::views::counted()`, you can let the compiler decide, which type is best for a range defined by a beginning and a size.

9.3.1 Lifetime Dependencies of Spans

Due to their reference semantics you can only iterate over spans while the underlying sequence of values exists. However, the iterators are not bound to the lifetime of the spans they are created from.

Iterators of spans do not refer to the span that they are created from. Instead, they refer to the underlying range directly. Therefore, a span is a borrowed range. This means that you can use the iterators even when the span no longer exists (the sequence of elements still has to exist, of course). However, note that iterators can still dangle when the underlying range is no longer there.

9.3.2 Performance of Spans

Spans are designed with best performance as a goal. Internally, they use just a raw pointer to the sequence of elements. However, raw pointers expect elements to be stored sequentially in one chunk of memory (otherwise raw pointers could to compute the position of elements when they jump back and forth). For this reason, spans require that the elements are stored in contiguous memory.

With this requirement, spans can provide the best performance of all types of views. Spans do not need any allocation and do not come with any indirection. The only overhead on using a span is the overhead of constructing it.[1]

Spans check whether a referred sequence has its elements in contiguous memory using concepts at compile time. When a sequence is initialized or a new sequence is assigned, iterators have to satisfy the concept `std::contiguous_iterator` and containers or ranges have to satisfy the two concepts `std::ranges::contiguous_range` and `std::ranges::sized_range`.

[1] Thanks to Barry Revzin for pointing this out.

Because spans are just a pointer and a size internally, it is very cheap to copy them. For this reason, you should prefer passing spans by value instead of passing them by `const` reference (which, however, is a problem for generic functions dealing with both containers and views).

Type Erasure

The fact that spans perform element access with raw pointers to the memory means that a span type erases the information of where the elements are stored. A span to the elements of a vector has the same type as a span to the elements of an array (provided they have the same extent):

```
std::array arr{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5};

std::span<int> vecSpanDyn{vec};
std::span<int> arrSpanDyn{arr};
std::same_as<decltype(arrSpanDyn), decltype(vecSpanDyn)>    // true
```

However, note that class template argument deduction for spans deduces a fixed extent from arrays and a dynamic extent from vectors. That means:

```
std::array arr{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5};
std::span arrSpan{arr};            // deduces std::span<int, 5>
std::span vecSpan{vec};            // deduces std::span<int>
std::span<int, 5> vecSpan5{vec};

std::same_as<decltype(arrSpan), decltype(vecSpan)>          // false
std::same_as<decltype(arrSpan), decltype(vecSpan5)>         // true
```

Spans vs. Subranges

The requirement for contiguous storage of the elements is the major difference to subranges, which are also introduced with C++20. Internally, subranges still use iterators and can therefore refer to all types of containers and ranges. However, that may lead to significantly more overhead.

In addition, spans do not require iterator support for the type they refer to. You can pass any type that provides a `data()` member for access to a sequence of elements.

9.3.3 `const` Correctness of Spans

Spans are views that have reference semantics. In that sense, they behave like pointers: if a span is `const`, it does not automatically mean that the elements the span refers to are `const`.

This means that you have write access to the elements of a `const` span (provided the elements are not `const`):

```
std::array a1{1, 2, 3, 4, 5, 6,7 ,8, 9, 10};
std::array a2{0, 8, 15};

const std::span<int> sp1{a1};   // span/view is const
std::span<const int> sp2{a1};   // elements are const

sp1[0] = 42;     // OK
sp2[0] = 42;     // ERROR

sp1 = a2;        // ERROR
sp2 = a2;        // OK
```

Note that as long as the elements of a `std::span<>` are not declared to be `const`, a couple of operations provide write access to the elements even for `const` spans where you might not expect it (following the rules for ordinary containers):

- `operator[]`, `first()`, `last()`
- `data()`
- `begin()`, `end()`, `rbegin()`, `rend()`
- `std::cbegin()`, `std::cend()`, `std::crbegin()`, `std::crend()`
- `std::ranges::cbegin()`, `std::ranges::cend()`, `std::ranges::crbegin()`, `std::ranges::crend()`

Yes, all the `c*` functions that were designed to ensure that the elements are `const` **are broken** by `std::span`.

For example:

```
template<typename T>
void modifyElemsOfConstColl (const T& coll)
{
  coll[0] = {};   // OK for spans, ERROR for regular containers

  auto ptr = coll.data();
  *ptr = {};      // OK for spans, ERROR for regular containers

  for (auto pos = std::cbegin(coll); pos != std::cend(coll); ++pos) {
    *pos = {};    // OK for spans, ERROR for regular containers
  }
}

std::array arr{1, 2, 3, 4, 5, 6, 7 ,8, 9, 10};

modifyElemsOfConstColl(arr);                  // ERROR: elements are const
modifyElemsOfConstColl(std::span{arr});      // OOPS: compiles and modifies elements of a1
```

The problem here is not that `std::span` is broken; the problem is that functions like `std::cbegin()` and `std::ranges::cbegin()` are currently broken for collections with reference semantics (such as views).

To ensure that a function only takes sequences where you cannot modify the elements that way, you can require that `begin()` for a `const` container returns an iterator to `const` elements:

```
template<typename T>
void ensureReadOnlyElemAccess (const T& coll)
requires std::is_const_v<std::remove_reference_t<decltype(*coll.begin())>>
{
  ...
}
```

The fact that even `std::cbegin()` and `std::ranges::cbegin()` provide write access is something under discussion after standardizing C++20. The whole point of providing `cbegin()` and `cend()` is to ensure that elements cannot be modified when iterating over them. Originally, spans did provide members for types `const_iterator`, `cbegin()`, and `cend()` to ensure that you could not modify elements. However, at the last minute before C++20 was finished, it turned out that `std::cbegin()` still iterates over mutable elements (and `std::ranges::cbegin()` has the same problem). However, instead of fixing `std::cbegin()` and `std::ranges::cbegin()`, the members for `const` iterators in spans were removed (see `http://wg21.link/lwg3320`), which made the problem even worse because in C++20 there is now no easy way to do a read-only iteration over a span unless the elements are `const`. It appears that `std::ranges::cbegin()` will be fixed with C++23 (see `http://wg21.link/p2278`). However, `std::cbegin()` will still be broken (sigh).

9.3.4 Using Spans as Parameters in Generic Code

As written, you can implement a generic function just for all spans with the following declaration:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp);
```

This even works for spans with dynamic extent, because they just use the special value `std::dynamic_extent` as size.

Therefore, in the implementation, you can deal with the difference between fixed and dynamic extent as follows:

lib/spanprint.hpp

```
#ifndef SPANPRINT_HPP
#define SPANPRINT_HPP

#include <iostream>
#include <span>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  std::cout << '[' << sp.size() << " elems";
  if constexpr (Sz == std::dynamic_extent) {
    std::cout << " (dynamic)";
  }
  else {
    std::cout << " (fixed)";
  }
  std::cout << ':';
  for (const auto& elem : sp) {
    std::cout << ' ' << elem;
  }
  std::cout << "]\n";
}

#endif // SPANPRINT_HPP
```

You might also consider declaring the element type to be `const`:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<const T, Sz> sp);
```

However, in this case, you cannot pass spans with non-`const` element types. Conversions from a non-`const` type to a `const` do not propagate to templates (for good reasons).

Lack of type deduction and conversions also disable passing an ordinary container such as a vector to this function. You need either an explicit type specification or an explicit conversion:

```
printSpan(vec);                          // ERROR: template type deduction doesn't work
printSpan(std::span{vec});               // OK
printSpan<int, std::dynamic_extent>(vec); // OK (provided it is a vector of ints)
```

For this reason, `std::span<>` should not be used as a vocabulary type for generic functions that deal with sequences of elements stored in contiguous memory.

For performance reasons, you might do something like this:[2]

```
template<typename E>
void processSpan(std::span<typename E>) {
  ...      // span specific implementation
}

template<typename T>
void print(const T& t) {
  if constexpr (std::ranges::contiguous_range<T> t) {
    processSpan<std::ranges::range_value_t<T>>(t);
  }
  else {
    ...    // generic implementations for all containers/ranges
  }
}
```

2 Thanks to Arthur O'Dwyer for pointing this out.

Using Spans as Ranges and Views

Because spans are *views*, they satisfy the concept `std::ranges::view`.[3] This means that spans can be used in all algorithms and functions for ranges and views. In the chapter that discusses the details of all views, we list the view-specific properties of spans.

One property of spans is that they are borrowed ranges, meaning that the lifetime of iterators does not depend on the lifetime of the span. For that reason, we can use a temporary span as a range in algorithms that yield iterators to it:

```
std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
auto pos1 = std::ranges::find(std::span{coll.data(), 3}, 42);  // OK
std::cout << *pos1 << '\n';
```

However, note that it is an error if the span refers to a temporary object. The following code compiles even though an iterator to a destroyed temporary object is returned:

```
auto pos2 = std::ranges::find(std::span{getData().data(), 3}, 42);
std::cout << *pos2 << '\n';          // runtime ERROR
```

9.4 Span Operations

This section describes the types and operations of spans in detail.

9.4.1 Span Operations and Member Types Overview

Table *Span operations* lists all operations that are provided for spans.

It is worth noting all the operations that are not supported:

- Comparisons (not even ==)
- `swap()`
- `assign()`
- `at()`
- I/O operators
- `cbegin()`, `cend()`, `crbegin()`, `crend()`
- Hashing
- Tuple-like API for structured bindings

That is, spans are neither containers (in the traditional C++ STL sense) nor regular types.

Regarding static members and member types, spans provide the usual members of containers (except `const_iterator`) plus two special ones: `element_type` and `extent` (see table *Static and type members of spans*).

Note that `std::value_type` is ***not*** the specified element type (as the `value_type` for `std::array` and several other types usually is). It is the element type with `const` and `volatile` removed.

[3] The original C++20 standard did require that views have to have a default constructor, which is not the case for fixed spans. However, that requirement was removed later with `http://wg21.link/P2325R3`.

Operation	**Effect**
constructors	Creates or copies a span
destructor	Destroys a span
`=`	Assigns a new sequence of values
`empty()`	Returns whether the span is empty
`size()`	Returns the number of elements
`size_bytes()`	Returns the size of memory used for all elements
`[]`	Accesses an element
`front()`, `back()`	Accesses the first or last element
`begin()`, `end()`	Provides iterator support (no `const_iterator` support)
`rbegin()`, `rend()`	Provides constant reverse iterator support
`first(`*n*`)`	Returns a sub-span with dynamic extent of the first *n* elements
`first<`*n*`>()`	Returns a sub-span with fixed extent of the first *n* elements
`last(`*n*`)`	Returns a sub-span with dynamic extent of the last *n* elements
`last<`*n*`>()`	Returns a sub-span with fixed extent of the last *n* elements
`subspan(`*offs*`)`	Returns a sub-span with dynamic extent skipping the first *offs* elements
`subspan(`*offs*`, `*n*`)`	Returns a sub-span with dynamic extent of *n* after *offs* elements
`subspan<`*offs*`>()`	Returns a sub-span with **same** extent skipping the first *offs* elements
`subspan<`*offs*`, `*n*`>()`	Returns a sub-span with fixed extent of *n* after *offs* elements
`data()`	Returns a raw pointer to the elements
`as_bytes()`	Returns the memory of the elements as a span of read-only `std::bytes`
`as_writable_bytes()`	Returns the memory of the elements as a span of writable `std::bytes`

Table 9.1. Span operations

Member	**Effect**
`extent`	Number of elements or `std::dynamic_extent` if size varies
`size_type`	Type of `extent` (always `std::size_t`)
`difference_type`	Difference type for pointers to elements (always `std::difference_type`)
`element_type`	Specified type of the elements
`pointer`	Type of a pointer to the elements
`const_pointer`	Type of a pointer for read-only access to the elements
`reference`	Type of a reference to the elements
`const_reference`	Type of a reference for read-only access to the elements
`iterator`	Type of an iterator to the elements
`reverse_iterator`	Type of a reverse iterator to the elements
`value_type`	Type of elements without `const` or `volatile`

Table 9.2. Static and type members of spans

9.4.2 Constructors

For spans, the default constructor is provided only if it has a dynamic extent or the extent is 0:

```
std::span<int> sp0a;        // OK
std::span<int, 0> sp0b;     // OK
std::span<int, 5> sp0c;     // compile-time ERROR
```

If this initialization is valid, `size()` is `0` and `data()` is `nullptr`.

In principle, you can initialize a span for an array, a beginning with a sentinel (end iterator), and a beginning with a size. The latter is used by the view `std::views::counted(`*beg*`,` *sz*`)` if *beg* points to elements in contiguous memory. Class template argument deduction is also supported.

This means that when initializing the span with a raw array or `std::array<>`, a span with a fixed extent is deduced (unless only the element type is specified):

```
int a[10] {};
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

std::span sp1a{a};                                  // std::span<int, 10>
std::span sp1b{arr};                                // std::span<int, 15>
std::span<int> sp1c{arr};                           // std::span<int>
std::span sp1d{arr.begin() + 5, 5};                 // std::span<int>
auto sp1e = std::views::counted(arr.data() + 5, 5); // std::span<int>
```

When initializing the span with a `std::vector<>`, a span with a dynamic extent is deduced unless a size is specified explicitly:

```
std::vector vec{1, 2, 3, 4, 5};

std::span sp2a{vec};                                // std::span<int>
std::span<int> sp2b{vec};                           // std::span<int>
std::span<int, 2> sp2c{vec};                        // std::span<int, 2>
std::span<int, std::dynamic_extent> sp2d{vec};      // std::span<int>
std::span<int, 2> sp2e{vec.data() + 2, 2};          // std::span<int, 2>
std::span sp2f{vec.begin() + 2, 2};                 // std::span<int>
auto sp2g = std::views::counted(vec.data() + 2, 2); // std::span<int>
```

If you initialize a span from an rvalue (temporary object), the elements have to be `const`:

```
std::span sp3a{getArrayOfInt()};                   // ERROR: rvalue and not const
std::span<int> sp3b{getArrayOfInt()};              // ERROR: rvalue and not const
std::span<const int> sp3c{getArrayOfInt()};        // OK
std::span sp3d{getArrayOfConstInt()};              // OK

std::span sp3e{getVectorOfInt()};                  // ERROR: rvalue and not const
std::span<int> sp3f{getVectorOfInt()};             // ERROR: rvalue and not const
std::span<const int> sp3g{getVectorOfInt()};       // OK
```

Initializing a span with a returned temporary collection can result in a fatal runtime error. For example, you should never use a range-based `for` loop to iterate over a span that is initialized directly:

```
for (auto elem : std::span{getCollOfConst()}) ...              // fatal runtime error

for (auto elem : std::span{getCollOfConst()}.last(2)) ...      // fatal runtime error

for (auto elem : std::span<const Type>{getColl()}) ...         // fatal runtime error
```

The problem is that the range-based `for` loop causes undefined behavior when iterating over a reference returned for a temporary object because the temporary object is destroyed before the loop starts to iterate internally. This is a bug that the C++ standards committee has not been willing to fix for years now (see `http://wg21.link/p2012`).

As a workaround, you can use the new range-based `for` loop with initialization:

```
for (auto&& coll = getCollOfConst(); auto elem : std::span{coll}) ...      // OK
```

Regardless of whether you initialize a span with an iterator and a length or with two iterators that define a valid range, the iterators have to refer to elements in contiguous memory (satisfy the concept `std::contiguous_iterator`):

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp6a{vec};                        // OK, refers to all elements
std::span<int> sp6b{vec.data(), vec.size()};     // OK, refers to all elements
std::span<int> sp6c{vec.begin(), vec.end()};     // OK, refers to all elements
std::span<int> sp6d{vec.data(), 5};              // OK, refers to first 5 elements
std::span<int> sp6e{vec.begin()+2, 5};           // OK, refers to elements 3 to 7 (including)
std::list<int> lst{...};
std::span<int> sp6f{lst.begin(), lst.end()};     // compile-time ERROR
```

If the span has a fixed extent, it must match the number of elements in the passed range. In general, compilers cannot check this at compile time:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int, 10> sp7a{vec};                    // OK, refers to all elements
std::span<int, 5> sp7b{vec};                     // runtime ERROR (undefined behavior)
std::span<int, 20> sp7c{vec};                    // runtime ERROR (undefined behavior)
std::span<int, 5> sp7d{vec, 5};                  // compile-time ERROR
std::span<int, 5> sp7e{vec.begin(), 5};          // OK, refers to first 5 elements
std::span<int, 3> sp7f{vec.begin(), 5};          // runtime ERROR (undefined behavior)
std::span<int, 8> sp7g{vec.begin(), 5};          // runtime ERROR (undefined behavior)
std::span<int, 5> sp7h{vec.begin()};             // compile-time ERROR
```

You can also create and initialize a span directly with a raw array or a `std::array`. In that case, some runtime errors due to an invalid number of elements become a compile-time error:

```
int raw[10];
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp8a{raw};                   // OK, refers to all elements
std::span<int> sp8b{arr};                   // OK, refers to all elements
std::span<int, 5> sp8c{raw};                // compile-time ERROR
std::span<int, 5> sp8d{arr};                // compile-time ERROR
std::span<int, 5> sp8e{arr.data(), 5};      // OK
```

That is: either you pass a container with sequential elements as a whole, or you pass two arguments to specify the initial range of elements. In any case, the number of elements must match a specified fixed extent.

Construction with Implicit Conversions

Spans have to have the element type of the elements of the sequence they refer to. Conversions (even implicit standard conversions) are not supported. However, additional qualifiers such as `const` are allowed. This also applies to copy constructors:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<const int> sp9a{vec};       // OK: element type with const
std::span<long> sp9b{vec};            // compile-time ERROR: invalid element type
std::span<int> sp9c{sp9a};            // compile-time ERROR: removing constness
std::span<const long> sp9d{sp9a};     // compile-time ERROR: different element type
```

To allow containers to refer to the elements of user-defined containers, these containers have to signal that they or their iterators require that all elements are in contiguous memory. For that, they have to fulfill the concept `contiguous_iterator`.

The constructors also allow the following type conversions between spans:

- Spans with a fixed extent convert to spans with the same fixed extent and additional qualifiers.
- Spans with a fixed extent convert to spans with dynamic extent.
- Spans with a dynamic extent convert to spans with a fixed extent, provided the current extent fits.

Using a conditional `explicit`, only the constructors of spans with fixed extent are `explicit`. In that case, *copy initialization* (using a =) is not possible if the initial value has to be converted:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> spanDyn{vec.begin(), 5};          // OK
std::span<int> spanDyn2 = {vec.begin(), 5};      // OK
std::span<int, 5> spanFix{vec.begin(), 5};       // OK
std::span<int, 5> spanFix2 = {vec.begin(), 5};   // ERROR
```

As a consequence, conversions are only implicitly supported when converting to a span with a dynamic or the same fixed extent:

```
void fooDyn(std::span<int>);
void fooFix(std::span<int, 5>);

fooDyn({vec.begin(), 5});                        // OK
fooDyn(spanDyn);                                 // OK
fooDyn(spanFix);                                 // OK
fooFix({vec.begin(), 5});                        // ERROR
fooFix(spanDyn);                                 // ERROR
fooFix(spanFix);                                 // OK

spanDyn = spanDyn;                               // OK
spanDyn = spanFix;                               // OK
spanFix = spanFix;                               // OK
spanFix = spanDyn;                               // ERROR
```

9.4.3 Operations for Sub-Spans

The member functions that create a sub-span can create spans with dynamic or fixed extent. Passing a size as a call parameter usually yields a span with dynamic extent. Passing a size as a template parameter usually yields a span with fixed extent. This is at least always the case for the member functions `first()` and `last()`:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};

auto sp1 = spDyn.first(2);      // first 2 elems with dynamic extent
auto sp2 = spDyn.last(2);       // last 2 elems with dynamic extent
auto sp3 = spDyn.first<2>();    // first 2 elems with fixed extent
auto sp4 = spDyn.last<2>();     // last 2 elems with fixed extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto sp5 = spFix.first(2);      // first 2 elems with dynamic extent
auto sp6 = spFix.last(2);       // last 2 elems with dynamic extent
auto sp7 = spFix.first<2>();    // first 2 elems with fixed extent
auto sp8 = spFix.last<2>();     // last 2 elems with fixed extent
```

However, for `subspan()`, the result may sometimes be surprising. Passing call parameters always yields spans with dynamic extent:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};
```

```
auto s1 = spDyn.subspan(2);                       // 3rd to last elem with dynamic extent
auto s2 = spDyn.subspan(2, 2);                    // 3rd to 4th elem with dynamic extent
auto s3 = spDyn.subspan(2, std::dynamic_extent);  // 3rd to last with dynamic extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto s4 = spFix.subspan(2);                       // 3rd to last elem with dynamic extent
auto s5 = spFix.subspan(2, 2);                    // 3rd to 4th elem with dynamic extent
auto s6 = spFix.subspan(2, std::dynamic_extent);  // 3rd to last with dynamic extent
```

However, when passing template parameters, the result might be different than what you expect:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};

auto s1 = spDyn.subspan<2>();                       // 3rd to last with dynamic extent
auto s2 = spDyn.subspan<2, 2>();                    // 3rd to 4th with fixed extent
auto s3 = spDyn.subspan<2, std::dynamic_extent>(); // 3rd to last with dynamic extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto s4 = spFix.subspan<2>();                       // 3rd to last with fixed extent
auto s5 = spFix.subspan<2, 2>();                    // 3rd to 4th with fixed extent
auto s6 = spFix.subspan<2, std::dynamic_extent>(); // 3rd to last with fixed extent
```

9.5 Afternotes

Spans were motivated as `array_views` by Lukasz Mendakiewicz and Herb Sutter in http://wg21.link/n3851 and first proposed by Neil MacIntosh in http://wg21.link/p0122r0. The finally accepted wording was formulated by Neil MacIntosh and Stephan T. Lavavej in http://wg21.link/p0122r7.

Several modifications were added later on, such as removing all comparison operators as proposed by Tony Van Eerd in http://wg21.link/P1085R2, and removing `const_iterator` support with the resolution of http://wg21.link/lwg3320.

After C++20 was released, the definition of views changed so that spans are now always views (see http://wg21.link/p2325r3).

Chapter 10

Formatted Output

The C++ IOStream library provides only inconvenient and limited ways to support formatted output (specifying a field width, fill characters, etc.). For that reason, formatted output often still uses functions like `sprintf()`.

C++20 introduces a new library for formatted output, which is described in this chapter. It allows convenient specification of formatting attributes and is extensible.

10.1 Formatted Output by Example

Before we go into details, let us look at some motivating examples.

10.1.1 Using `std::format()`

The basic function of the formatting library for application programmers is `std::format()`. It allows them to combine a formatting string to be filled with values of passed arguments according to the formatting specified inside a pair of braces. A simple example is to use the values of each passed argument:

```
#include <format>

std::string str{"hello"};
...
std::cout << std::format("String '{}' has {} chars\n", str, str.size());
```

The function `std::format()`, defined in `<format>`, takes a format string (a string literal, string, or string view known at compile time) in which `{...}` stands for the value of the next argument (here, using the default formatting of its type). It yields a `std::string`, for which it allocates memory.

The output of this first example is as follows:

```
String 'hello' has 5 chars
```

An optional integral value right after the opening braces specifies the index of the argument so that you can process the arguments in different order or use them multiple times. For example:

```
std::cout << std::format("{1} is the size of string '{0}'\n", str, str.size());
```

has the following output:

```
5 is the size of string 'hello'
```

Note that you do not have to explicitly specify the type of the argument. This means that you can easily use `std::format()` in generic code. Consider the following example:

```
void print2(const auto& arg1, const auto& arg2)
{
  std::cout << std::format("args: {} and {}\n", arg1, arg2);
}
```

If you call this function as follows:

```
print2(7.7, true);
print2("character:", '?');
```

it has the following output:

```
args: 7.7 and true
args: character: and ?
```

The formatting even works for user-defined types if formatted output is supported. Formatted output of the chrono library is one example. A call such as the following

```
print2(std::chrono::system_clock::now(), std::chrono::seconds{13});
```

might have the following output:

```
args: 2022-06-19 08:46:45.3881410 and 13s
```

For your own types you need a *formatter*, which is described later.

Inside the placeholder for formatting after a colon, you can specify details of the formatting of the passed argument. For example, you can define a field width:

```
std::format("{:7}", 42)          // yields "     42"
std::format("{:7}", 42.0)        // yields "     42"
std::format("{:7}", 'x')         // yields "x      "
std::format("{:7}", true)        // yields "true   "
```

Note that the different types have different default alignments. Note also that for a `bool`, the values `false` and `true` are printed instead of `0` and `1` as for iostream output with the operator `<<`.

You can also specify the alignment explicitly (`<` for left, `^` for center, and `>` for right) and specify a fill character:

```
std::format("{:*<7}", 42)        // yields "42*****"
std::format("{:*>7}", 42)        // yields "*****42"
std::format("{:*^7}", 42)        // yields "**42***"
```

Several additional formatting specifications are possible to force a specific notation, a specific precision (or limit strings to a certain size), fill characters, or a positive sign:

```
std::format("{:7.2f} Euro", 42.0)  // yields "  42.00 Euro"
std::format("{:7.4}", "corner")    // yields "corn   "
```

By using the position of the argument, we can print a value in multiple forms. For example:

```
std::cout << std::format("'{0}' has value {0:02X} {0:+4d} {0:03o}\n", '?');
std::cout << std::format("'{0}' has value {0:02X} {0:+4d} {0:03o}\n", 'y');
```

print the hexadecimal, decimal, and octal value of `'?'` and `'y'` using the actual character set, which might look as follows:

```
'?' has value 3F  +63 077
'y' has value 79 +121 171
```

10.1.2 Using `std::format_to_n()`

The implementation of `std::format()` has a pretty good performance when compared with other ways of formatting. However, memory has to be allocated for the resulting string. To save time, you can use `std::format_to_n()`, which writes to a preallocated array of characters. You have to specify both the buffer to write to and its size. For example:

```
char buffer[64];
...
auto ret = std::format_to_n(buffer, std::size(buffer) - 1,
                            "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0';
```

or:

```
std::array<char, 64> buffer;
...
auto ret = std::format_to_n(buffer.begin(), buffer.size() - 1,
                            "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0';  // write trailing null terminator
```

Note that `std::format_to_n()` does ***not*** write a trailing null terminator. However, the return value holds all information to deal with this. It is a data structure of type `std::format_to_n_result` that has two members:

- out for the position of the first character not written
- `size` for the number of characters that would have been written without truncating them to the passed size.

For that reason, we store a null terminator at the end, where `ret.out` points to. Note that we only pass `buffer.size()-1` to `std::format_to_n()` to ensure we have memory for the trailing null terminator:

```
auto ret = std::format_to_n(buffer.begin(), buffer.size() - 1, ...);
*(ret.out) = '\0';
```

Alternatively, we can initialize the buffer with `{}` to ensure that all characters are initialized with the null terminator.

It is not an error if the size does not fit for the value. In that case, the written value is simply cut. For example:

```
std::array<char, 5> mem{};
std::format_to_n(mem.data(), mem.size()-1, "{}", 123456.78);
```

```
std::cout << mem.data() << '\n';
```

has the following output:

```
1234
```

10.1.3 Using `std::format_to()`

The formatting library also provides `std::format_to()`, which writes an unlimited number of characters for formatted output. Using this function for limited memory is a risk because if the value needs too much memory, you have undefined behavior. However, by using an output stream buffer iterator, you can safely use it to write directly to a stream:

```
std::format_to(std::ostreambuf_iterator<char>{std::cout},
               "String '{}' has {} chars\n", str, str.size());
```

In general, `std::format_to()` takes any output iterator for characters. For example, you can also use a back inserter to append the characters to a string:

```
std::string s;
std::format_to(std::back_inserter(s),
               "String '{}' has {} chars\n", str, str.size());
```

The helper function `std::back_inserter()` creates an object that calls `push_back()` for each character. Note that the implementation of `std::format_to()` can recognize that back insert iterators are passed and write multiple characters for certain containers at once so that we still have good performance.[1]

10.1.4 Using `std::formatted_size()`

To know in advance how many characters would be written by a formatted output (without writing any), you can use `std::formatted_size()`. For example:

```
auto sz = std::formatted_size("String '{}' has {} chars\n", str, str.size());
```

That would allow you to reserve enough memory or double check that the reserved memory fits.

10.2 Performance of the Formatting Library

One reason to still use `sprintf()` was that it has a significantly better performance than using output string streams or `std::to_string()`. The formatting library was designed with the goal of doing this better. It should format at least as fast as `sprintf()` or even faster.

Current (draft) implementations show that an equal or even better performance is possible. Rough measurements show that

- `std::format()` should be as fast as or even better than `sprintf()`
- `std::format_to()` and `std::format_to_n()` should be even better

[1] Thanks to Victor Zverovich for pointing this out.

The fact that compilers can usually check format strings at compile time does help a lot. It helps to avoid mistakes with formatting as well as to get a significantly better performance.

However, the performance ultimately depends on the quality of the implementation of the formatting libraries on your specific platform.[2] Therefore, you should measure the performance yourself. The program *format/formatperf.cpp* might give you some hints abut the situation on your platform.

10.2.1 Using std::vformat() and vformat_to()

To support this goal, an important fix was adopted for the formatting library after C++20 was standardized (see http://wg21.link/p2216r3). With this fix, the functions `std::format()`, `std::format_to()`, and `std::format_to_n()` ***require*** that the format string is a ***compile-time*** value. You have to pass a string literal or a `constexpr` string. For example:

```
const char* fmt1 = "{}\n";                        // runtime format string
std::cout << std::format(fmt1, 42);               // compile-time ERROR: runtime format

constexpr const char* fmt2 = "{}\n";              // compile-time format string
std::cout << std::format(fmt2, 42);               // OK
```

As a consequence, invalid format specifications become compile-time errors:

```
std::cout << std::format("{:7.2f}\n", 42);        // compile-time ERROR: invalid format

constexpr const char* fmt2 = "{:7.2f}\n";         // compile-time format string
std::cout << std::format(fmt2, 42);               // compile-time ERROR: invalid format
```

Of course, applications sometimes have to compute formatting details at runtime (such as computing the best width according to the passed values). In that case, you have to use `std::vformat()` or `std::vformat_to()` and pass all arguments with `std::make_format_args()` to these functions:

```
const char* fmt3 = "{} {}\n";                                   // runtime format
std::cout << std::vformat(fmt3, std::make_format_args(42, 1.7));    // OK
```

If a runtime format string is used and the formatting is not valid for the passed argument, the call throws a runtime error of type `std::format_error`:

```
const char* fmt4 = "{:7.2f}\n";
std::cout << std::vformat(fmt4, std::make_format_args(42));  // runtime ERROR:
                                               // throws std::format_error exception
```

[2] For example, while writing this book, for Visual C++, the option `/utf-8` improves the formatting performance significantly.

10.3 Formatted Output in Detail

This section describes the syntax of formatting in detail.

10.3.1 General Format of Format Strings

The general way to specify the formatting of arguments is to pass a *format string* that can have *replacement fields* specified by `{...}` and characters. All other characters are printed as they are. To print the characters `{` and `}`, use `{{` and `}}`.

For example:

```
std::format("With format {{}}: {}", 42);    // yields "With format {}: 42"
```

The replacement fields may have an index to specify the argument and a format specifier after a colon:

- **`{}`**
  Use the next argument with its default formatting
- **`{`*n*`}`**
  Use the *n*-th argument (first argument has index `0`) with its default formatting
- **`{:`*fmt*`}`**
  Use the next argument formatted according to *fmt*
- **`{`*n*`:`*fmt*`}`**
  Use the *n*-th argument formatted according to *fmt*

You can either have none of the arguments specified by an index or all of them:

```
std::format("{}: {}", key, value);          // OK
std::format("{1}: {0}", value, key);        // OK
std::format("{}: {} or {0}", value, key);   // ERROR
```

Additional arguments are ignored.

The syntax of the format specifier depends on the type of the passed argument.

- For arithmetic types, strings, and raw pointers, a standard format has been defined by the formatting library itself.
- In addition, C++20 specifies a standard formatting of chrono types (durations, timepoints, and calendrical types).

10.3.2 Standard Format Specifiers

Standard format specifiers have the following format (each specifier is optional):

fill align sign **#** **0** *width* .*prec* **L** *type*

- *fill* is the character to fill the value up to *width* (default: space). It can only be specified if *align* is also specified.
- *align* is
  - **<** left-aligned
  - **>** right-aligned

  – ^ centered

  The default alignment depends on the type.
- *sign* is
  – **-** only the negative sign for negative values (default)
  – **+** positive or negative sign
  – *space* negative sign or space
- **#** switches to an *alternative form* of some notations:
  – It writes a prefix such as `0b`, `0`, and `0x` for the binary, octal, and hexadecimal notation of integral values.
  – It forces the floating-point notation to always write a dot.
- **0** in front of *width* pads arithmetic values with zeros.
- *width* specifies a minimum field width.
- *prec* after a dot specifies the precision:
  – For floating-point types, it specifies how many digits are printed after the dot or in total (depending on the notation).
  – For string types, it specifies the maximum number of characters processed from the string.
- **L** activates locale-dependent formatting (this may impact the format of arithmetic types and `bool`)
- *type* specifies the general notation of the formatting. This allows you to print characters as integral values (or vice versa) or choose the general notation of floating-point values.

10.3.3 Width, Precision, and Fill Characters

For all values printed, a positive integral value after the colon (without a leading dot) specifies a minimal field width for the output of the value as a whole (including sign etc.). It can be used together with a specification of the alignment: For example:

```
std::format("{:7}", 42);        // yields "     42"
std::format("{:7}", "hi");      // yields "hi     "
std::format("{:^7}", "hi");     // yields "  hi   "
std::format("{:>7}", "hi");     // yields "     hi"
```

You can also specify padding zeros and a fill character. The padding `0` can be used only for arithmetic types (except `char` and `bool`) and is ignored if an alignment is specified:

```
std::format("{:07}", 42);       // yields "0000042"
std::format("{:^07}", 42);      // yields "  42   "
std::format("{:>07}", -1);      // yields "     -1"
```

A padding `0` is different from the general fill character that can be specified immediately after the colon (in front of the alignment):

```
std::format("{:^07}", 42);      // yields "  42   "
std::format("{:0^7}", 42);      // yields "0042000"
std::format("{:07}", "hi");     // invalid (padding 0 not allowed for strings)
std::format("{:0<7}", "hi");    // yields "hi00000"
```

The precision is used for floating-point types and strings:

- For floating-point types, you can specify a different precision than the usual default 6:

  ```
  std::format("{}", 0.12345678);          // yields "0.12345678"
  std::format("{:.5}", 0.12345678);       // yields "0.12346"
  std::format("{:10.5}", 0.12345678);     // yields "   0.12346"
  std::format("{:^10.5}", 0.12345678);    // yields " 0.12346   "
  ```

  Note that depending on the floating-point notation, the precision might apply to the value as a whole or to the digits after the dot.
- For strings, you can use it to specify a maximum number of characters:

  ```
  std::format("{}", "counterproductive");          // yields "counterproductive"
  std::format("{:20}", "counterproductive");       // yields "counterproductive   "
  std::format("{:.7}", "counterproductive");       // yields "counter"
  std::format("{:20.7}", "counterproductive");     // yields "counter             "
  std::format("{:^20.7}", "counterproductive");    // yields "      counter       "
  ```

Note that width and precision can be parameters themselves. For example, the following code:

```
int width = 10;
int precision = 2;
for (double val : {1.0, 12.345678, -777.7}) {
  std::cout << std::format("{:+{}.{}f}\n", val, width, precision);
}
```

has the following output:

```
     +1.00
    +12.35
   -777.70
```

Here, we specify at runtime that we have a minimum field width of 10 with two digits after the dot (using the `fixed` notation).

10.3.4 Format/Type Specifiers

By specifying a *format* or *type specifier*, you can force various notations for integral types, floating-point types, and raw pointers.

Specifiers for Integral Types

Table *Formatting Options for integral types* lists the possible formatting type options for integral types (including `bool` and `char`).[3]

[3] According to `http://wg21.link/lwg3648`, the support for specifier `c` for `bool` is probably an error and will be removed.

Spec.	**42**	**'@'**	**true**	**Meaning**
none	42	@	true	Default format
d	42	64	1	Decimal notation
b / B	101010	1000000	1	Binary notation
#b	0b101010	0b1000000	0b1	Binary notation with prefix
#B	0B101010	0B1000000	0B1	Binary notation with prefix
o	52	100	1	Octal notation
x	2a	40	1	Hexadecimal notation
X	2A	40	1	Hexadecimal notation
#x	0x2a	0x40	0x1	Hexadecimal notation with prefix
#X	0X2A	0X40	0X1	Hexadecimal notation with prefix
c	*	@	'\1'	As character with the value
s	*invalid*	*invalid*	true	bool as string

Table 10.1. Formatting options for integral types

For example:

```
std::cout << std::format("{:#b} {:#b} {:#b}\n", 42, '@', true);
```

will print:

```
0b101010 0b1000000 0b1
```

Note the following:

- The default notations are:
  - d (decimal) for integer types
  - c (as character) for character types
  - s (as string) for type bool
- If L is specified after the notation, the locale-dependent character sequence for Boolean values and the locale-dependent thousands separator and decimal point characters for arithmetic values are used.

Specifiers for Floating-Point Types

Table *Formatting options for floating-point types* lists the possible formatting type options for floating-point types.

For example:

```
std::cout << std::format("{0} {0:#} {0:#g} {0:e}\n", -1.0);
```

will print:

```
-1 -1. -1.00000 -1.000000e+00
```

Note that passing the integer -1 instead would be a formatting error.

Spec.	-1.0	0.0009765625	1785856.0	Meaning
none	-1	0.0009765625	1.785856e+06	Default format
#	-1.	0.0009765625	1.785856e+06	Forces decimal point
f / F	-1.000000	0.000977	1785856.000000	Fixed notation (default precision after dot: 6)
g	-1	0.000976562	1.78586e+06	Fixed or exponential notation (default full precision: 6)
G	-1	0.000976562	1.78586E+06	Fixed or exponential notation (default full precision: 6)
#g	-1.00000	0.000976562	1.78586e+06	Fixed or exponential notation (forced dot and zeros)
#G	-1.00000	0.000976562	1.78586E+06	Fixed or exponential notation (forced dot and zeros)
e	-1.000000e+00	9.765625e-04	1.7858560e+06	Exponential notation (default precision after dot: 6)
E	-1.000000E+00	9.765625E-04	1.7858560E+06	Exponential notation (default precision after dot: 6)
a	-1p+0	1p-10	1.b4p+20	Hexadec. floating-point notation
A	-1P+0	1P-10	1.B4P+20	Hexadec. floating-point notation
#a	-1.p+0	1.p-10	1.b4p+20	Hexadec. floating-point notation
#A	-1.P+0	1.P-10	1.B4P+20	Hexadec. floating-point notation

Table 10.2. Formatting options for floating-point types

Specifiers for Strings

For string types, the default format specifier is **s**. However, you do not have to provide this specifier, because it is default. Note also that for strings, you can specify a certain precision, which is interpreted as the maximum number of characters used:

```
std::format("{}", "counter");          // yields "counter"
std::format("{:s}", "counter");        // yields "counter"
std::format("{:.5}", "counter");       // yields "count"
std::format("{:.5}", "hi");            // yields "hi"
```

Note that only the standard string types of the character types `char` and `wchar_t` are supported. There is no support for strings and sequences of the types `u8string` and `char8_t`, `u16string` and `char16_t`, or `u32string` and `char32_t`. In fact, the C++ standard library provides *formatters* for the following types:

- `char*` and `const char*`
- `const char[`*n*`]` (string literals)
- `std::string` and `std::basic_string<char,` *traits*, *allocator*`>`
- `std::string_view` and `std::basic_string_view<char,` *traits*`>`
- `wchar_t*` and `const wchar_t*`
- `const wchar_t[`*n*`]` (wide string literals)

- `std::wstring` and `std::basic_string<wchar_t,` *traits*`,` *allocator*`>`
- `std::wstring_view` and `std::basic_string_view<wchar_t,` *traits*`>`

Note that the format string and arguments for them must have the same character type:

```
auto ws1 = std::format("{}", L"K\u00F6ln");            // compile-time ERROR
std::wstring ws2 = std::format(L"{}", L"K\u00F6ln");   // OK
```

Specifiers for Pointers

For pointer types, the default format specifier is **p**, which usually writes the address in hexadecimal notation with the prefix `0x`. On platforms that have no type `uintptr_t` the format is implementation-defined:

```
void* ptr = ... ;
std::format("{}", ptr)          // usually yields a value such as 0x7ff688ee64
std::format("{:p}", ptr)        // usually yields a value such as 0x7ff688ee64
```

Note that only the following pointer types are supported:

- `void*` and `const void*`
- `std::nullptr_t`

Thus, you can either pass `nullptr` or a raw pointer, which you have to cast to type `(const) void*`:

```
int i = 42;
std::format("{}", &i)                                   // compile-time error
std::format("{}", static_cast<void*>(&i))               // OK (e.g., 0x7ff688ee64)
std::format("{:p}", static_cast<void*>(&i))             // OK (e.g., 0x7ff688ee64)
std::format("{}", static_cast<const void*>("hi"))       // OK (e.g., 0x7ff688ee64)
std::format("{}", nullptr)                              // OK (usually 0x0)
std::format("{:p}", nullptr)                            // OK (usually 0x0)
```

10.4 Internationalization

If `L` is specified for a format, the locale-specific notation is used:

- For `bool`, the locale strings on `std::numpunct::truename` and `std::numpunct::falsename` are used.
- For integral values, the locale-dependent thousands separator character is used.
- For floating-point values, the locale-dependent decimal point and thousands separator characters are used.
- For several notations of types in the chrono library (durations, timepoints, etc), their locale-specific formats are used.

To activate the locale-specific notation, you also have to pass a locale to `std::format()`. For example:

```
// initialize a locale for "German in Germany":
#ifdef _MSC_VER
std::locale locG{"deu_deu.1252"};
#else
std::locale locG{"de_DE"};
#endif
```

```
// use it for formatting:
std::format(locG, "{0} {0:L}", 1000.7)   // yields 1000.7 1.000,7
```

See *format/formatgerman.cpp* for a complete example.

Note that the locale is used only if you use the locale specifier L. Without this, the default locale "C" is used, which uses the American formatting.

Alternatively, you can set the global locale and use the L specifier:

```
std::locale::global(locG);               // set German locale globally
std::format("{0} {0:L}", 1000.7)         // yields 1000.7 1.000,7
```

You might have to create your own locale (usually based on an existing locale with modified *facets*). For example:

format/formatbool.cpp

```
#include <iostream>
#include <locale>
#include <format>

// define facet for German bool names:
class GermanBoolNames : public std::numpunct_byname<char> {
 public:
  GermanBoolNames (const std::string& name)
   : std::numpunct_byname<char>(name) {
  }
 protected:
  virtual std::string do_truename() const {
    return "wahr";
  }
  virtual std::string do_falsename() const {
    return "falsch";
  }
};

int main()
{
  // create locale with German bool names:
  std::locale locBool{std::cin.getloc(),
                      new GermanBoolNames{""}};

  // use locale to print Boolean values:
  std::cout << std::format(locBool, "{0} {0:L}\n", false);  //false falsch
}
```

The program has the following output:

```
false falsch
```

To print values with wide-character strings (which is an issue especially with Visual C++), both the format string and the arguments have to be wide-character strings. For example:

```
std::wstring city = L"K\u00F6ln";              // Köln
auto ws1 = std::format("{}", city);            // compile-time ERROR
std::wstring ws2 = std::format(L"{}", city);   // OK: ws2 is std::wstring
std::wcout << ws2 << '\n';                     // OK
```

Strings of types `char8_t` (UTF-8 characters), `char16_t`, and `char32_t` are not supported, yet.

10.5 Error Handling

Ideally, C++ compilers should detect bugs at compile time rather than at runtime. Because string literals are known at compile time, C++ can check for format violations when string literals are used as format strings and does so in `std::format()`:[4]

```
std::format("{:d}", 42)                          // OK
std::format("{:s}", 42)                          // compile-time ERROR
```

If you pass a format string that has already been initialized or computed, the formatting library handles format errors as follows:

- `std::format()`, `std::format_to()`, and `format_to_n()` take only format strings known at compile time:
  - String literals
  - `constexpr` character pointers
  - Compile-time strings that can be converted to a compile-time string view
- To use format strings computed at runtime, use
  - `std::vformat()`
  - `std::vformat_to()`
- For `std::formatted_size()`, you can only use format strings known at compile time.

For example:

```
const char* fmt1 = "{:d}";                       // runtime format string
std::format(fmt1, 42);                           // compile-time ERROR
std::vformat(fmt1, std::make_format_args(42));   // OK

constexpr const char* fmt2 = "{:d}";             // compile-time format string
std::format(fmt2, 42);                           // OK
```

Using `fmt1` does not compile because the passed argument is not a compile-time string and `std::format()` is used. However, using `fmt1` with `std::vformat()` works fine (but you have to convert all arguments with `std::make_format_args()`). Using `fmt2` does compile when you pass it to `std::format()` because it is initialized as a compile-time string.

[4] The requirement to check format strings at compile time was proposed with http://wg21.link/p2216r3 and accepted as a fix against C++20 after C++20 was standardized.

If you want to use multiple arguments with `std::vformat()`, you have to pass them all to one call of `std::make_format_args()`:

```
const char* fmt3 = "{} {}";
std::vformat(fmt3, std::make_format_args(x, y))
```

If a format failure is detected at runtime, an exception of type `std::`**`format_error`** is thrown. This new standard exception type is derived from `std::runtime_error` and offers the usual API of standard exceptions to initialize the exception with a string for the error message you get by calling `what()`.

For example:

```
try {
  const char* fmt4 = "{:s}";
  std::vformat(fmt4, std::make_format_args(42))   // throws std::format_error
}
catch (const std::format_error& e) {
  std::cerr << "FORMATTING EXCEPTION: " << e.what() << std::endl;
}
```

10.6 User-Defined Formatted Output

The formatting library can define formatting for user-defined types. What you need is a *formatter*, which is pretty straightforward to implement.

10.6.1 Basic Formatter API

A *formatter* is a specialization of the class template `std::formatter<>` for your type(s). Inside the formatter, two member functions have to be defined:

- `parse()` to implement how to parse the format string specifiers for your type
- `format()` to perform the actual formatting for an object/value of your type

Let us look at a first minimal example (we will improve it step by step) that specifies how to format an object/value that has a fixed value. Assume the type is defined like this (see *format/always40.hpp*):

```
class Always40 {
 public:
  int getValue() const {
    return 40;
  }
};
```

For this type, we can define a first formatter (which we should definitely improve) as follows:

format/formatalways40.hpp

```
#include "always40.hpp"
#include <format>
#include <iostream>

template<>
struct std::formatter<Always40>
{
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    return ctx.begin();     // return position of } (hopefully there)
  }

  // format by always writing its value:
  auto format(const Always40& obj, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{}", obj.getValue());
  }
};
```

This is already good enough so that the following works:

```
Always40 val;
std::cout << std::format("Value: {}\n", val);
```

```
std::cout << std::format("Twice: {0} {0}\n", val);
```

The output would be:

```
Value: 40
Twice: 40 40
```

We define the formatter as a specialization of type `std::formatter<>` for our type `Always40`:

```
template<>
struct std::formatter<Always40>
{
  ...
};
```

Because we only have public members, we use `struct` instead of `class`.

Parsing the Format String

In `parse()`, we implement the function to parse the format string:

```
// parse the format string for this type:
constexpr auto parse(std::format_parse_context& ctx) {
  return ctx.begin();      // return position of } (hopefully there)
}
```

The function takes a `std::format_parse_context`, which provides an API to iterate over the remaining characters of the passed format string. `ctx.begin()` points to the first character of the format specifier for the value to be parsed or the } if there is no specifier:

- If the format string is `"Value: {:7.2f}"`
  `ctx.begin()` points to: `"7.2f}"`
- If the format string is `"Twice: {0} {0}"`
  `ctx.begin()` points to: `"} {0}"`
  when called for the first time
- If the format string is `"{}\n"`
  `ctx.begin()` points to: `"}\n"`

There is also a `ctx.end()`, which points to the end of the whole format string. This means that the opening { was already parsed and you have to parse all characters until the corresponding closing }.

For the format string `"Val: {1:_>20}cm \n"`, `ctx.begin()` is the position of the `_` and `ctx.end()` is the end of the whole format string after the `\n`. The task of `parse()` is to parse the specified format of the passed argument, which means that you have to parse only the characters `_>20`, and then return the position of the end of the format specifier which is the trailing } behind the character `0`.

In our implementation, we do not support any format specifier yet. Therefore, we simply return the position of the first character we get, which will work only if the next character really is the } (dealing with character before the closing } is the first thing we have to improve). Calling `std::format()` with any specified format character will not work:

```
Always40 val;
std::format("'{:7}", val)   // ERROR
```

Note that the parse() member function should be constexpr to support compile-time computing of the format string. This means that the code has to accept all restrictions of constexpr functions (which were relaxed with C++20).

However, you can see how this API allows programmers to parse whatever format we have specified for their types. This is used, for example, to support formatted output of the chrono library. Of course, we should follow the conventions of the standard specifiers to avoid programmer confusion.

Performing the Formatting

In format(), we implement the function to format the passed value:

```
// format by always writing its value:
auto format(const Always40& value, std::format_context& ctx) const {
  return std::format_to(ctx.out(), "{}", value.getValue());
}
```

The function takes two parameters:

- Our value passed as an argument to std::format() (or similar functions)
- A std::format_context, which provides the API to write the resulting characters of the formatting (according to the parsed format)

The most important function of the format context is out(), which yields an object you can pass to std::format_to() to write the actual characters of the formatting. The function has to return the new position for further output, which is returned by std::format_to().

Note that the format() member function of a formatter should be const. According to the original C++20 standard, that was not required (see http://wg21.link/lwg3636 for details).

10.6.2 Improved Parsing

Let us improve the example we saw previously. First, we should ensure that the parser deals with the format specifiers in a better way:

- We should take care of all characters up to the closing }.
- We should throw an exception when illegal formatting arguments are specified.
- We should deal with valid formatting arguments (such as a specified field width).

Let us look at all of these topics by looking at an improved version of the previous formatter (this time dealing with a type that always has the value 41):

format/formatalways41.hpp

```
#include "always41.hpp"
#include <format>

template<>
class std::formatter<Always41>
{
  int width = 0;   // specified width of the field
 public:
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    auto pos = ctx.begin();
    while (pos != ctx.end() && *pos != '}') {
      if (*pos < '0' || *pos > '9') {
        throw std::format_error{std::format("invalid format '{}'", *pos)};
      }
      width = width * 10 + *pos - '0';   // new digit for the width
      ++pos;
    }
    return pos;                      // return position of }
  }

  // format by always writing its value:
  auto format(const Always41& obj, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{:{}}", obj.getValue(), width);
  }
};
```

Our formatter now has a member to store the specified field width:

```
template<>
class std::formatter<Always41>
{
  int width = 0;   // specified width of the field
  ...
};
```

The field width is initialized with `0`, but can be specified by the format string.

The parser now has a loop that processes all characters up to the trailing `}`:

```
constexpr auto parse(std::format_parse_context& ctx) {
  auto pos = ctx.begin();
  while (pos != ctx.end() && *pos != '}') {
    ...
    ++pos;
  }
  return pos;                      // return position of }
}
```

Note that the loop has to check for both whether there is still a character and whether it is the trailing `}`, because the programmer calling `std::format()` might have forgotten the trailing `}`.

Inside the loop, we multiply the current width with the integral value of the digit character:

```
width = width * 10 + *pos - '0';   // new digit for the width
```

If the character is not a digit, we throw a `std::format` exception initialized with `std::format()`:

```
if (*pos < '0' || *pos > '9') {
  throw std::format_error{std::format("invalid format '{}'", *pos)};
}
```

Note that we cannot use `std::isdigit()` here because it is not a function we could call at compile time.

You can test the formatter as follows: *format/always41.cpp*

The program has the following output:

```
41
Value: 41
Twice: 41 41
With width: '      41'
Format Error: invalid format 'f'
```

The value is right-aligned because this is the default alignment for integral values.

10.6.3 Using Standard Formatters for User-Defined Formatters

We can still improve the formatter implemented above:

- We can allow alignment specifiers.
- We can support fill characters.

Fortunately, we do not have to implement the complete parsing ourselves. Instead, we can use the standard formatters to benefit from the format specifiers they support. Actually, there are two approaches for doing that:

- You can ***delegate*** the works to a local standard formatter.
- You can ***inherit*** from a standard formatter.

Delegating Formatting to Standard Formatters

To delegate formatting to standard formatters, you have to

- Declare a local standard formatter
- Let the `parse()` function delegate the work to the standard formatter
- Let the `format()` function delegate the work to the standard formatter

In general, this should look as follows:

format/formatalways42ok.hpp

```
#include "always42.hpp"
#include <format>

// *** formatter for type Always42:
template<>
struct std::formatter<Always42>
{
  // use a standard int formatter that does the work:
```

```
  std::formatter<int> f;

  // delegate parsing to the standard formatter:
  constexpr auto parse(std::format_parse_context& ctx) {
    return f.parse(ctx);
  }

  // delegate formatting of the value to the standard formatter:
  auto format(const Always42& obj, std::format_context& ctx) const {
    return f.format(obj.getValue(), ctx);
  }
};
```

As usual, we declare a specialization of `std::formatter<>` for type `Always42`. However, this time, we use a local standard formatter for type `int` that does the work. We delegate both the parsing and the formatting to it. In fact, we extract the value from our type with `getValue()` and use the standard `int` formatter to do the rest of the formatting.

We can test the formatter with the following program:

format/always42.cpp

```
#include "always42.hpp"
#include "formatalways42.hpp"
#include <iostream>

int main()
{
  try {
    Always42 val;
    std::cout << val.getValue() << '\n';
    std::cout << std::format("Value: {}\n", val);
    std::cout << std::format("Twice: {0} {0}\n", val);
    std::cout << std::format("With width: '{:7}'\n", val);
    std::cout << std::format("With all:   '{:.^7}'\n", val);
  }
  catch (std::format_error& e) {
    std::cerr << "Format Error: " << e.what() << std::endl;
  }
}
```

The program has the following output:

```
42
Value: 42
Twice: 42 42
With width: '     42'
With all:   '..42...'
```

Note that the value is still right-aligned by default because that is the default alignment for `int`.

Note also that in practice, you might need some modifications for this code, which is discussed later in detail:

- Declaring `format()` as `const` might not compile unless the formatter is declared as `mutable`.
- Declaring `parse()` as `constexpr` might not compile.

Inheriting From Standard Formatters

Usually, it is even enough to derive from a standard formatter so that the formatter member and its `parse()` function are implicitly available:

format/formatalways42inherit.hpp

```
#include "always42.hpp"
#include <format>

// *** formatter for type Always42:
// - use standard int formatter
template<>
struct std::formatter<Always42> : std::formatter<int>
{
  auto format(const Always42& obj, std::format_context& ctx) {
    // delegate formatting of the value to the standard formatter:
    return std::formatter<int>::format(obj.getValue(), ctx);
  }
};
```

However, note that in practice, you might also need some modifications for this code:

- Declaring `format()` as `const` might not compile.

Using Standard Formatters in Practice

In practice, there are some issues with what was standardized with C++20 so that afterwards some things had to be clarified:

- C++20 as originally standardized did not require that the `format()` member function of a formatter should be `const` (see `http://wg21.link/lwg3636` for details). To support implementations of the C++ standard library that do not declare `format()` as a `const` member function, you have to either declare it as a non-`const` function or declare the local formatter as `mutable`.[5]
- Existing implementations might not yet support compile-time parsing with a `parse()` member function that is `constexpr` because compile-time parsing was added after C++20 was standardized (see `http://wg21.link/p2216r3`). In that case, we cannot delegate compile-time parsing to a standard formatter.

[5] Thanks to Arthur O'Dwyer for pointing this out.

As a consequence, in practice, the formatter for type `Always42` might have to look as follows:

format/formatalways42.hpp

```cpp
#include "always42.hpp"
#include <format>

// *** formatter for type Always42:
template<>
struct std::formatter<Always42>
{
  // use a standard int formatter that does the work:
#if __cpp_lib_format < 202106
  mutable      // in case the standard formatters have a non-const format()
#endif
  std::formatter<int> f;

  // delegate parsing to the standard int formatter:
#if __cpp_lib_format >= 202106
  constexpr    // in case standard formatters don't support constexpr parse() yet
#endif
  auto parse(std::format_parse_context& ctx) {
    return f.parse(ctx);
  }

  // delegate formatting of the int value to the standard int formatter:
  auto format(const Always42& obj, std::format_context& ctx) const {
    return f.format(obj.getValue(), ctx);
  }
};
```

As you can see, the code

- Declares `parse()` only with `constexpr` if the corresponding fix was adopted
- Might declare the local formatter with `mutable` so that the `const format()` member function can call a standard `format()` function that is not `const`

For both, the implementation uses a feature test macro that signals that compile-time parsing is supported (expecting that its adoptions also make the `format()` member functions of the standard formatters `const`).

10.6.4 Using Standard Formatters for Strings

If you have more complicated types to format, one common approach is to create a string and then use the standard formatter for strings (`std::string` or `std::string_view` if only string literals are used).

For example, we can define an enumeration type and a formatter for it as follows:

format/color.hpp

```
#include <format>
#include <string>

enum class Color { red, green, blue };

// *** formatter for enum type Color:
template<>
struct std::formatter<Color> : public std::formatter<std::string>
{
  auto format(Color c, format_context& ctx) const {
    // initialize a string for the value:
    std::string value;
    switch (c) {
      using enum Color;
      case red:
        value = "red";
        break;
      case green:
        value = "green";
        break;
      case blue:
        value = "blue";
        break;
      default:
        value = std::format("Color{}", static_cast<int>(c));
        break;
    }
    // and delegate the rest of formatting to the string formatter:
    return std::formatter<std::string>::format(value, ctx);
  }
};
```

By inheriting the formatter from the formatter for strings, we inherit its `parse()` functions, which means that we support all format specifiers strings have. In the `format()` function, we then perform the mapping to a string and then let the standard formatter do the rest of the formatting.

We can use the formatter as follows:

format/color.cpp

```
#include "color.hpp"
#include <iostream>
#include <string>
#include <format>
```

```
int main()
{
  for (auto val : {Color::red, Color::green, Color::blue, Color{13}}) {
    // use user-provided formatter for enum Color:
    std::cout << std::format("Color {:_>8} has value {:02}\n",
                             val, static_cast<int>(val));
  }
}
```

The program has the following output:

```
Color _____red has value 00
Color ___green has value 01
Color ____blue has value 02
Color _Color13 has value 13
```

This approach usually works fine if you do not introduce your own format specifiers. If only string literals are used as possible values, you could even use the formatter for `std::string_view` instead.

10.7 Afternotes

Formatted output was first proposed by Victor Zverovich and Lee Howes in http://wg21.link/p0645r0. The finally accepted wording was formulated by Victor Zverovich in http://wg21.link/p0645r10.

After C++20 was standardized, several fixes against C++20 were accepted. The most important was that format strings are checked and must be known at compile time, as formulated by Victor Zverovich in http://wg21.link/p2216r3. Others are http://wg21.link/p2372r3 (fixing locale handling of chrono formatters) and http://wg21.link/p2418r2 (format arguments become universal/forwarding references).

Chapter 11

Dates and Timezones for `<chrono>`

C++11 introduced the chrono library with basic support for durations and timepoints. This allowed you to specify and deal with durations of different units and timepoints of different clocks. However, there was no support yet for high-level durations and timepoints such as dates (days, months, and years), weekdays, and for dealing with different timezones.

C++20 extends the existing chrono library with support for dates, timezones, and several other features. This extension is described in this chapter.[1]

11.1 Overview by Example

Before we go into details, let us look at some motivating examples.

11.1.1 Scheduling a Meeting on the 5th of Every Month

Consider a program where we want to to iterate over all months in a year to schedule a meeting on the 5th of every month. The program can be written as follows:

lib/chrono1.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;        // shortcut for std::chrono
  using namespace std::literals;      // for h, min, y suffixes

```

[1] Many thanks to Howard Hinnant, the author of this library, for his incredible support in writing this chapter. He provided fast and elaborated answers, several code examples, and even some particular wording (with permission, not marked as a quote to keep the chapter easy to read).

```
  // for each 5th of all months of 2021:
  chr::year_month_day first = 2021y / 1 / 5;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << ":\n";

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

The program has an output like the following:

```
2021-01-05:
 We meet on Tuesday 01/05/21 at 18:30
2021-02-05:
 We meet on Friday 02/05/21 at 18:30
2021-03-05:
 We meet on Friday 03/05/21 at 18:30
2021-04-05:
 We meet on Monday 04/05/21 at 18:30
2021-05-05:
 We meet on Wednesday 05/05/21 at 18:30
...
2021-11-05:
 We meet on Friday 11/05/21 at 18:30
2021-12-05:
 We meet on Sunday 12/05/21 at 18:30
```

Let us go through this program step by step.

Namespace Declarations

We start with the namespace and `using` declarations to make the use of the chrono library a little more convenient:

- First we introduce `chr::` as a shortcut for the standard namespace of the chrono library `std::chrono`:

  ```
  namespace chr = std::chrono;          // shortcut for std::chrono
  ```

  To make examples more readable, from time to time I will use just `chr::` instead of `std::chrono::`.
- Then, we ensure that we can use literal suffixes such as `s`, `min`, `h`, and `y` (the latter is new since C++20).

  ```
  using namespace std::literals;        // for min, h, y suffixes
  ```

To avoid any qualification, you could use a `using` declaration instead:

```
using namespace std::chrono;          // skip chrono namespace qualification
```

As usual, you should limit the scope of such a using declaration to avoid unwanted side effects.

Calendrical Type `year_month_day`

Data flow starts with the initialization of the start date `first` of type `std::chrono::year_month_day`:

```
chr::year_month_day first = 2021y / 1 / 5;
```

The type `year_month_day` is a calendrical type that provides attributes for all three fields of a date so that it is easy to deal with the year, the month, and the day of a particular date.

Because we want to iterate over all fifth days of each month, we initialize the object with January 5, 2021, using operator **/** to combine a year with a value for the month and a value for the day as follows:

- First, we create the year as an object of the type `std::chrono::year`. Here, we use the new standard literal **`y`**:

  ```
  2021y
  ```

  To have this literal available, we have to provide one of the following `using` declarations:

  ```
  using std::literals;                // enable all standard literals
  using std::chrono::literals;        // enable all standard chrono literals
  using namespace std::chrono;        // enable all standard chrono literals
  using namespace std;                // enable all standard literals
  ```

  Without these literals we would need something like the following:

  ```
  std::chrono::year{2021}
  ```

- Then, we use operator **/** to combine the `std::chrono::year` with an integral value, which creates an object of type `std::chrono::year_month`.

  Because the first operand is a `year`, it is clear that the second operand must be a month. You cannot specify a day here.

- Finally, we use operator **/** again to combine the `std::chrono::year_month` object with an integral value to create a `std::chrono::year_month_day`.

This initialization for calendrical types is type-safe and requires only the specification the type of the first operand.

Because the operator already yields the right type, we could also declare `first` with `auto`. Without the namespace declarations, we would have to write:

```
auto first = std::chrono::year{2021} / 1 / 5;
```

With the chrono literals available, we could simply write:

```
auto first = 2021y/1/5;
```

Other Ways to Initialize Calendrical Types

There are other ways to initialize a calendrical type like `year_month_day`:

```
auto d1 = std::chrono::years{2021}/1/5;       // January 5, 2021
auto d2 = std::chrono:month{1}/5/2021;        // January 5, 2021
auto d3 = std::chrono:day{5}/1/2021;          // January 5, 2021
```

That is, the type of the first argument for operator **/** specifies how to interpret the other arguments.

With the chrono literals available, we could simply write:

```
using namespace std::literals;
auto d4 = 2021y/1/5;                    // January 5, 2021
auto d5 = 5/1/2021;                     // January 5, 2021
```

There is no standard suffix for a month, but we have predefined standard objects:

```
auto d6 = std::chrono::January / 5 / 2021;    // January 5, 2021
```

With the corresponding using declaration, we could even write just the following:

```
using namespace std::chrono;
auto d6 = January/5/2021;               // January 5, 2021
```

In all cases, we initialize an object of type `std::chrono::year_month_day`.

New Duration Types

In our example, we then call a loop to iterate over all months of the year:

```
for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
  ...
}
```

The type `std::chrono::months` is a new duration type that represents a month. You can use it for all calendrical types to deal with the specific month of a date, such as adding one month as we have done here:

```
std::chrono::year_month_day d = ... ;
d += chr::months{1};                          // add one month
```

However, note that when we use it for ordinary durations and timepoints, the type `months` represents the ***average duration*** of a month, which is `30.436875` days. Therefore, with ordinary timepoints, you should use `months` and `years` with care.

Output Operators for All Chrono Types

Inside the loop, we print the current date:

```
std::cout << d << '\n';
```

Since C++20, there is an output operator defined for more or less all possible chrono types.

```
std::chrono::year_month_day d = ... ;
std::cout << "d: " << d << '\n';
```

This makes it easy to just print any chrono value. However, the output does not always fit specific needs. For type `year_month_day`, the output format is what we as programmers would expect for such a type: *year-month-day*. For example:

```
2021-01-05
```

The other default output formats usually use a slash as the separator, which is documented here.

For user-defined output, the chrono library also supports the new library for formatted output. We use that later to print out the timepoint `tp`:

```
std::cout << std::format("We meet on {:%D at %R}\n", tp);
```

Formatting specifiers such as `%A` for the weekday, `%D` for the date, and `%R` for the time (hour and minute) correspond with the specifiers for the C function `strftime()` and the POSIX `date` command, so that the output might, for example, look as follows:

```
We meet on 10/05/21 at 18:30
```

Locale-specific output is also supported. The details of formatted output for chrono types is described later in detail.

Combining Dates and Times

To initialize `tp`, we combine the days of the loop with a specific time of the day:

```
auto tp{sys_days{d} + 18h + 30min};
```

To combine dates and times we have to use timepoints and durations. A calendrical type such as `std::chrono::year_month_day` is not a timepoint. Therefore, we first convert the `year_month_day` value to a `time_point<>` object:

```
std::chrono::year_month_day d = ... ;
std::chrono::sys_days{d}     // convert to a time_point
```

The type `std::chrono::sys_days` is a new shortcut for system timepoints with the granularity of days. It is equivalent to: `std::chrono::time_point<std::chrono::system_clock, std::chrono::days>`.

By adding some durations (18 hours and 30 minutes), we compute a new value, which—as usual in the chrono library—has a type with a granularity that is good enough for the result of the computation. Because we combine days with hours and minutes, the resulting type is a system timepoint with the granularity of minutes. However, we do not have to know the type. Just using `auto` works fine.

To specify the type of `tp` more explicitly, we could also declare it as follows:

- As system timepoint without specifying its granularity:

  ```
  chr::sys_time tp{chr::sys_days{d} + 18h + 30min};
  ```

  Thanks to class template argument deduction, the template parameter for the granularity is deduced.
- As system timepoint with a specified granularity:

  ```
  chr::sys_time<chr::minutes> tp{chr::sys_days{d} + 18h + 30min};
  ```

  In this case, the initial value must not be more fine grained than the specified type or you have to use rounding functions.
- As system timepoint convenient type for seconds as granularity:

  ```
  chr::sys_seconds tp{chr::sys_days{d} + 18h + 30min};
  ```

  Again, the initial value must not be more fine grained than the specified type or you have to use rounding functions.

In all these cases, the default output operator prints the timepoint according to the specified format as described above. For example:

```
We meet on 10/05/21 at 18:30
```

Note that when dealing with system time, by default the output is UTC.

A more fine-grained timepoint would also use whatever is necessary to output the exact value (such as milliseconds). See later.

11.1.2 Scheduling a Meeting on the Last Day of Every Month

Let us modify the first example program by iterating over the last days of each month. The way you can do that is as follows:

lib/chrono2.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;       // shortcut for std::chrono
  using namespace std::literals;     // for h, min, y suffixes

  // for each last of all months of 2021:
  auto first = 2021y / 1 / chr::last;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << ":\n";

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

All we modified was the initialization of `first`. We declare it with type `auto` and initialize it with `std::chrono::last` as day:

```
auto first = 2021y / 1 / chr::last;
```

The type `std::chrono::last` not only represents the last day of a month, it also has the effect that `first` has a different type: `std::chrono::year_month_day_last`. The effect is that by adding one month, the day of the date gets adjusted. In fact, the type just always holds the *last day*. The conversion to the numerical day happens when we output the date and specify a corresponding output format.

As a result, the output changes to the following:

```
2021/Jan/last:
 We meet on Sunday 01/31/21 at 18:30
2021/Feb/last:
 We meet on Sunday 02/28/21 at 18:30
2021/Mar/last:
 We meet on Wednesday 03/31/21 at 18:30
2021/Apr/last:
 We meet on Friday 04/30/21 at 18:30
2021/May/last:
 We meet on Monday 05/31/21 at 18:30
...
2021/Nov/last:
 We meet on Tuesday 11/30/21 at 18:30
2021/Dec/last:
 We meet on Friday 12/31/21 at 18:30
```

As you can see, the default output format of `year_month_day_last` uses `last` and slashes instead of dashes as a separator (only `year_month_day` uses hyphens in its default output format). For example:

```
2021/Jan/last
```

You could still declare `first` as `std::chrono::year_month_day`:

```
// for each last days of all months of 2021:
std::chrono::year_month_day first = 2021y / 1 / chr::last;
for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
  // print out the date:
  std::cout << d << ":\n";

  // init and print 18:30 UTC of those days:
  auto tp{chr::sys_days{d} + 18h + 30min};
  std::cout << std::format(" We meet on {:%D at %R}\n", tp);
}
```

However, this would cause the following output:

```
2021-01-31:
 We meet on Sunday 01/31/21 at 18:30
2021-02-31 is not a valid date:
 We meet on Wednesday 03/03/21 at 18:30
2021-03-31:
 We meet on Wednesday 03/31/21 at 18:30
2021-04-31 is not a valid date:
 We meet on Saturday 05/01/21 at 18:30
2021-05-31:
 We meet on Monday 05/31/21 at 18:30
...
```

Because the type of `first` stores the ***numeric*** value of the day, initialized with the last day of January, we iterate over the 31st day of each month. If such a day does not exist, the default output format prints that this is an invalid date, while `std::format()` even performs a bad computation.

A way to deal with such situations is to check whether a date is valid and implement what to do then. For example:

lib/chrono3.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;        // shortcut for std::chrono
  using namespace std::literals;      // for h, min, y suffixes
  // for each last of all months of 2021:
  chr::year_month_day first = 2021y / 1 / 31;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    if (d.ok()) {
      std::cout << d << ":\n";
    }
    else {
      // for months not having the 31st use the 1st of the next month:
      auto d1 = d.year() / d.month() / 1 + chr::months{1};
      std::cout << d << ":\n";
    }

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

By using the member function `ok()`, we adjust an invalid date to take the first day of the next month. We get the following output:

```
2021-01-31:
 We meet on Sunday 01/31/21 at 18:30
2021-02-31 is not a valid date:
 We meet on Wednesday 03/03/21 at 18:30
2021-03-31:
 We meet on Wednesday 03/31/21 at 18:30
2021-04-31 is not a valid date:
 We meet on Saturday 05/01/21 at 18:30
2021-05-31:
 We meet on Monday 05/31/21 at 18:30
...
2021-11-31 is not a valid date:
 We meet on Wednesday 12/01/21 at 18:30
2021-12-31:
 We meet on Friday 12/31/21 at 18:30
```

11.1.3 Scheduling a Meeting Every First Monday

In a similar way, we can schedule a meeting on each first Monday of a month:

lib/chrono4.cpp

```
#include <chrono>
#include <iostream>

int main()
```

```
{
  namespace chr = std::chrono;         // shortcut for std::chrono
  using namespace std::literals;       // for min, h, y suffixes

  // for each first Monday of all months of 2021:
  auto first = 2021y / 1 / chr::Monday[1];
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << '\n';

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

Again, `first` has a special type, `std::chrono::year_month_weekday`, which represents a weekday in a month of a year:

The default output format signals that with a specific format. However, the formatted output works fine:

```
2021/Jan/Mon[1]
 We meet on Monday 01/04/21 at 18:30
2021/Feb/Mon[1]
 We meet on Monday 02/01/21 at 18:30
2021/Mar/Mon[1]
 We meet on Monday 03/01/21 at 18:30
2021/Apr/Mon[1]
 We meet on Monday 04/05/21 at 18:30
2021/May/Mon[1]
 We meet on Monday 05/03/21 at 18:30
...
2021/Nov/Mon[1]
 We meet on Monday 11/01/21 at 18:30
2021/Dec/Mon[1]
 We meet on Monday 12/06/21 at 18:30
```

Calendrical Types for Weekdays

This time, we initialize `first` as an object of the calendrical type `std::chrono::year_month_weekday` and initialize it with the first Monday of January of 2021:

```
auto first = 2021y / 1 / chr::Monday[1];
```

Again, we use `operator/` to combine different date fields. However, this time, types for weekdays come into play:

- First, we call `2021y / 1` to combine a `std::chrono::year` with an integral value to create a `std::chrono::year_month`.

- Then, we call `operator[]` for `std::chrono::Monday`, which is a standard object of type `std::chrono::weekday`, to create an object of type `std::chrono::weekday_indexed` representing the *n*th weekday.
- Finally, `operator/` is used to combine the `std::chrono::year_month` with the object of type `std::chrono::weekday_indexed`, which creates a `std::chrono::year_month_weekday` object.

Therefore, a fully specified declaration would look as follows:

```
std::chrono::year_month_weekday first = 2021y / 1 / std::chrono::Monday[1];
```

Again, note that the default output format of `year_month_weekday` uses slashes instead of dashes as a separator (only `year_month_day` uses hyphens in its default output format). For example:

```
2021/Jan/Mon[1]
```

11.1.4 Using Different Timezones

Let us modify the first example program to bring timezones into play. In fact, we want the program to iterate over all first Mondays of each month in a year and schedule a meeting in different timezones.

For this we need the following modifications:

- Iterate over all months of the **current year**
- Schedule the meeting at 18:30 our **local time**
- Print the meeting time using **other timezones**

The program can now be written as follows:

lib/chronotz.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono;          // shortcut for std::chrono
  using namespace std::literals;        // for min, h, y suffixes

  try {
    // initialize today as current local date:
    auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
    chr::year_month_day today{chr::floor<chr::days>(localNow)};
    std::cout << "today: " << today << '\n';

    // for each first Monday of all months of the current year:
    auto first = today.year() / 1 / chr::Monday[1];
    for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
      // print out the date:
      std::cout << d << '\n';
```

```
        // init and print 18:30 local time of those days:
        auto tp{chr::local_days{d} + 18h + 30min};                        // no timezone
        std::cout << "  tp:      " << tp << '\n';

        // apply this local time to the current timezone:
        chr::zoned_time timeLocal{chr::current_zone(), tp};               // local time
        std::cout << "  local:   " << timeLocal << '\n';

        // print out date with other timezones:
        chr::zoned_time timeUkraine{"Europe/Kiev", timeLocal};            // Ukraine time
        chr::zoned_time timeUSWest{"America/Los_Angeles", timeLocal};     // Pacific time
        std::cout << "  Ukraine: " << timeUkraine << '\n';
        std::cout << "  Pacific: " << timeUSWest << '\n';
      }
    }
    catch (const std::exception& e) {
      std::cerr << "EXCEPTION: " << e.what() << '\n';
    }
  }
```

Starting the program in Europe in 2021 results in the following output:

```
today: 2021-03-29
2021/Jan/Mon[1]
  tp:      2021-01-04 18:30:00
  local:   2021-01-04 18:30:00 CET
  Ukraine: 2021-01-04 19:30:00 EET
  Pacific: 2021-01-04 09:30:00 PST
2021/Feb/Mon[1]
  tp:      2021-02-01 18:30:00
  local:   2021-02-01 18:30:00 CET
  Ukraine: 2021-02-01 19:30:00 EET
  Pacific: 2021-02-01 09:30:00 PST
2021/Mar/Mon[1]
  tp:      2021-03-01 18:30:00
  local:   2021-03-01 18:30:00 CET
  Ukraine: 2021-03-01 19:30:00 EET
  Pacific: 2021-03-01 09:30:00 PST
2021/Apr/Mon[1]
  tp:      2021-04-05 18:30:00
  local:   2021-04-05 18:30:00 CEST
  Ukraine: 2021-04-05 19:30:00 EEST
  Pacific: 2021-04-05 09:30:00 PDT
2021/May/Mon[1]
  tp:      2021-05-03 18:30:00
  local:   2021-05-03 18:30:00 CEST
  Ukraine: 2021-05-03 19:30:00 EEST
  Pacific: 2021-05-03 09:30:00 PDT
...
```

```
2021/Oct/Mon[1]
  tp:      2021-10-04 18:30:00
  local:   2021-10-04 18:30:00 CEST
  Ukraine: 2021-10-04 19:30:00 EEST
  Pacific: 2021-10-04 09:30:00 PDT
2021/Nov/Mon[1]
  tp:      2021-11-01 18:30:00
  local:   2021-11-01 18:30:00 CET
  Ukraine: 2021-11-01 19:30:00 EET
  Pacific: 2021-11-01 10:30:00 PDT
2021/Dec/Mon[1]
  tp:      2021-12-06 18:30:00
  local:   2021-12-06 18:30:00 CET
  Ukraine: 2021-12-06 19:30:00 EET
  Pacific: 2021-12-06 09:30:00 PST
```

Look at the output for October and November: in Los Angeles, the meeting is now scheduled at different times although the same timezone PDT is used. That happens because the source for the meeting time (central Europe) changed from summer to winter/standard time.

Again, let us go through the modifications of this program step by step.

Dealing with Today

The first new statement is the initialization of `today` as an object of type `year_month_day`:

```
auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
chr::year_month_day today = chr::floor<chr::days>(localNow);
```

Support for `std::chrono::system_clock::now()` has already been provided since C++11 and yields a `std::chrono::time_point<>` with the granularity of the system clock. This system clock uses UTC (since C++20, the `system_clock` is guaranteed to use *Unix time*, which is based on UTC). Therefore, we first have to adjust the current UTC time and date to the time and date of the current/local timezone, which `current_zone()->to_local()` does. Otherwise, our local date might not fit the UTC date (because we have already passed midnight but UTC has not, or the other way around).

Using `localNow` directly to initialize a `year_month_day` value does not compile, because we would "narrow" the value (lose the hours, minutes, seconds, and subseconds parts of the value). By using a convenience function such as `floor()` (available since C++17), we round the value down according to our requested granularity.

If you need the current date according to UTC, the following would be enough:

```
chr::year_month_day today = chr::floor<chr::days>(chr::system_clock::now());
```

Local Dates and Times

Again, we combine the days we iterate over with a specific time of day. However, this time, we convert each day to type `std::chrono::local_days` first:

```
auto tp{chr::local_days{d} + 18h + 30min};
```

The type `std::chrono::local_days` is a shortcut of `time_point<local_t, days>`. Here, the pseudo clock `std::chrono::local_t` is used, which means that we have a *local timepoint*, a timepoint with no associated timezone (not even UTC) yet.

The next statement combines the local timepoint with the current timezone so that we get a time-zone specific point of time of type `std::chrono::zoned_time<>`:

```
chr::zoned_time timeLocal{chr::current_zone(), tp};   // local time
```

Note that a timepoint is already associated with the system clock, which means that it already associates the time with UTC. Combining such a timepoint with a *different* timezone *converts* the time to a specific timezone.

The default output operators demonstrate the difference between a timepoint and a zoned time:

```
auto tpLocal{chr::local_days{d} + 18h + 30min};          // local timepoint
std::cout << "timepoint: " << tpLocal << '\n';

chr::zoned_time timeLocal{chr::current_zone(), tpLocal};  // apply to local timezone
std::cout << "zonedtime: " << timeLocal << '\n';
```

This code outputs, for example:

```
timepoint: 2021-01-04 18:30:00
zonedtime: 2021-01-04 18:30:00 CET
```

You see that the timepoint is printed without a timezone while the zoned time has a timezone. However, both outputs print the same time because we apply a local timepoint to our local timezone.

In fact, the difference between timepoints and zoned times is as follows:

- A *timepoint* ***may*** be associated with a defined epoch. It *may* define a unique point in time; however, it might also be associated with an undefined or pseudo epoch, for which the meaning is not clear yet until we combine it with a timezone.
- A *zoned time* ***is*** always associated with a timezone so that the epoch finally has a defined meaning. It always represents a unique point in time.

Look what happens when we use `std::chrono::sys_days` instead of `std::chrono::local_days`:

```
auto tpSys{chr::sys_days{d} + 18h + 30min};              // system timepoint
std::cout << "timepoint: " << tpSys << '\n';

chr::zoned_time timeSys{chr::current_zone(), tpSys};      // convert to local timezone
std::cout << "zonedtime: " << timeSys << '\n';
```

Here we use a system timepoint, which has an associated timezone UTC. While a local timepoint is ***applied*** to a timezone, as system timepoint is ***converted*** to a timezone. Therefore, when we run the program in a timezone with one hour difference we get the following output:

```
timepoint: 2021-01-04 18:30:00
zonedtime: 2021-01-04 19:30:00 CET
```

Using Other Timezones

Finally, we print the timepoint using different timezones, one for the Ukraine (the nation Russia waged a war on) passing Kiev and one for the Pacific timezone in North America passing Los Angeles:

```
chr::zoned_time timeUkraine{"Europe/Kiev", timeLocal};          // Ukraine time
chr::zoned_time timeUSWest{"America/Los_Angeles", timeLocal};   // Pacific time
std::cout << "  Ukraine: " << timeUkraine << '\n';
std::cout << "  Pacific: " << timeUSWest << '\n';
```

To specify a timezone, we have to use the official timezone names of the IANA timezone database, which is usually based on cities that represent the timezones. Timezone abbreviations such as PST may change over the year or apply to different timezones.

Using the default output operator of these objects adds the corresponding timezone abbreviation, whether it is in "winter time:"

```
local:   2021-01-04 18:30:00 CET
Ukraine: 2021-01-04 19:30:00 EET
Pacific: 2021-01-04 09:30:00 PST
```

or whether it is in "summer time:"

```
local:   2021-07-05 18:30:00 CEST
Ukraine: 2021-07-05 19:30:00 EEST
Pacific: 2021-07-05 09:30:00 PDT
```

Sometimes, some timezones are in summer time while others are not. For example, at the beginning of November we have daylight/summer time in the US but not in the Ukraine:

```
local:   2021-11-01 18:30:00 CET
Ukraine: 2021-11-01 19:30:00 EET
Pacific: 2021-11-01 10:30:00 PDT
```

When Timezones Are Not Supported

C++ can exist and be useful on small systems, even on a toaster. In that case, the availability of the IANA timezone database would cost far too much resources. Therefore, it is not required that the timezone database exists.

All timezone-specific calls throw an exception, if the timezone database is not supported. For this reason, you should catch exceptions when using timezones in a portable C++ program:

```
try {
  // initialize today as current local date:
  auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
  ...
}
catch (const std::exception& e) {
  std::cerr << "EXCEPTION: " << e.what() << '\n';   // IANA timezone DB missing
}
```

Note that for Visual C++, this also applies to older Windows systems, because the required OS support does not exist before Windows 10.

In addition, platforms may support the timezone API without using all of the IANA timezone database. For example, with a German Windows 11 installation, I get output such as `GMT-8` and `GMT-7` instead of `PST` and `PDT`.

11.2 Basic Chrono Concepts and Terminology

The chrono library was designed to be able to deal with the fact that timers and clocks might be different on different systems and improve in precision over time. To avoid having to introduce a new time type every 10 years or so, the basic goal established with C++11 was to provide a precision-neutral concept by separating duration and point of time ("timepoint"). C++20 extends these basic concepts with support for dates and timezones and a few other extensions.

As a result, the core of the chrono library consists of the following types or concepts:

- A ***duration*** of time is defined as a specific number of ticks over a time unit. One example is a duration such as "3 minutes" (3 ticks of a "minute"). Other examples are "42 milliseconds" or "86,400 seconds," which represents the duration of 1 day. This approach also allows the specification of something like "1.5 times a third of a second," where 1.5 is the number of ticks and "a third of a second" the time unit used.
- A ***timepoint*** is defined as combination of a duration and a beginning of time (the so-called ***epoch***).

  A typical example is a ***system timepoint*** that represents midnight on December 31, 2000. Because the system epoch is specified as Unix/POSIX birth, the timepoint would be defined as "946,684,800 seconds since January 1, 1970" (or 946,684,822 seconds when taking leap seconds into account, which some timepoints do).

  Note that the epoch might be unspecified or a pseudo epoch. For example, a ***local timepoint*** is associated with whatever local time we have. It still needs a certain value for the epoch, but the exact point in time it represents is not clear until we apply this timepoint to a specific timezone. Midnight on December 31 is a good example of that: the celebration and fireworks start at different times in the world depending on the timezone we are in.
- A ***clock*** is the object that defines the epoch of a timepoint. Thus, different clocks have different epochs. Each timepoint is parameterized by a clock.

  C++11 introduced two basic clocks (a `system_clock` to deal with the system time and a `steady_clock` for measurements and timers that should not be influenced by changes of the system clock). C++20 adds new clocks to deal with UTC timepoints (supporting leap seconds), GPS timepoints, TAI (international atomic time) timepoints, and timepoints of the filesystem.

  To deal with *local timepoints*, C++20 also adds a pseudo clock `local_t`, which is not bound to a specific epoch/origin.

  Operations that deal with multiple timepoints, such as processing the duration/difference between two timepoints, usually require use of the same epoch/clock. However, conversions between different clocks are possible.

  A clock (if not local) provides a convenience function `now()` to yield the current point in time.
- A ***calendrical*** type (introduced with C++20) allows us to deal with the attributes of calendars using the common terminology of days, months, and years. These types can be used to represent a single

attribute of a date (`day`, `month`, `year`, and `weekday`) and combinations of them (such as `year_month` or `year_month_day` for a full date).

Different symbols such as `Wednesday`, `November`, and `last` allow us to use common terminology for partial and full dates, such as "last Wednesday of November."

Fully specified calendrical dates (having a year, month, and day or specific weekday of the month) can be converted to or from timepoints using the system clock or the pseudo clock for the local time.

- A ***timezone*** (introduced with C++20) allows us to deal with the fact that simultaneous events result in different times when different timezones come into play. If we meet online at 18:30 UTC, the local time of the meeting would be significantly later in Asia but significantly earlier in America.

  Thus, timezones give timepoints a (different) meaning by applying or converting them to a different local time.
- A ***zoned time*** (introduced with C++20) is the combination of a timepoint with a timezone. It can be used to apply a local timepoint to a specific timezone (which might be the "current" timezone) or convert timepoints to different timezones.

  A zoned time can be seen as a date and time object, which is the combination of an epoch (the origin of a timepoint), a duration (the distance from the origin), and a timezone (to adjust the resulting time).

For all of these concepts and types, C++20 adds support for output (even formatted) and parsing. That way, you can decide whether you print date/time values in formats such as the following:

```
Nov/24/2011
24.11.2011
2011-11-24 16:30:00 UTC
Thursday, November 11, 2011
```

11.3 Basic Chrono Extensions with C++20

Before we discuss how to use the chrono library in detail, let us introduce all the basic types and symbols.

11.3.1 Duration Types

C++20 introduces additional duration types for days, weeks, months, and years. Table *Standard duration types since C++20* lists all duration types C++ now provides, together with the standard literal suffix you can use to create a value and the default output suffix used when printing values.

Be careful when using `std::chrono::months` and `std::chrono::years`. `months` and `years` represent the ***average*** duration of a month or year, which is a fractional day. The duration of an average year is computed by taking leap years into account:

- Every 4 years we have one more day (366 instead of 365 days)
- However, every 100 years we do not have one more day
- However, every 400 years we have one more day again

Therefore, the value is $400 * 365 + 100 - 4 + 1$ divided by 400. The duration of an average month is one twelfth of this.

Type	Defined as	Literal	Output Suffix
`nanoseconds`	1/1,000,000,000 seconds	`ns`	`ns`
`microseconds`	1,000 nanoseconds	`us`	`µs` or `us`
`milliseconds`	1,000 microseconds	`ms`	`ms`
`seconds`	1,000 milliseconds	`s`	`s`
`minutes`	60 seconds	`min`	`min`
`hours`	60 minutes	`h`	`h`
`days`	24 hours	`d`	`d`
`weeks`	7 days		`[604800]s`
`months`	30.436875 days		`[2629746]s`
`years`	365.2425 days	`y`	`[31556952]s`

Table 11.1. Standard duration types since C++20

11.3.2 Clocks

C++20 introduces a couple of new clocks (remember, clocks define the origin/epoch of a timepoint).

Table *Standard clock types since C++20* describes the names and meanings of all clocks that are provided now by the C++ standard library.

Type	Meaning	Epoch
`system_clock`	Associated with the clock of the system (since C++11)	UTC time
`utc_clock`	Clock for UTC time values	UTC time
`gps_clock`	Clock for GPS values	GPS time
`tai_clock`	Clock for international atomic time values	TAI time
`file_clock`	Clock for timepoints of the filesystem library	impl.spec.
`local_t`	Pseudo clock for *local timepoints*	open
`steady_clock`	Clock for measurements (since C++11)	impl.spec.
`high_resolution_clock`	(see text)	

Table 11.2. Standard clock types since C++20

The column **Epoch** specifies whether the clock specifies a unique point in time so that you always have a specific UTC time defined. This requires a stable specified epoch. For the `file_clock`, the epoch is system-specific but will be stable across multiple runs of a program. The epoch of the `steady_clock` might change from one run of your application to the next (e.g., when the system rebooted). For the pseudo clock `local_t`, the epoch is interpreted as "local time," which means you have to combine it with a timezone to know which point in time it represents.

The C++ standard also provides a `high_resolution_clock` since C++11. However, its use can introduce subtle issues when porting code from one platform to another. In practice, `high_resolution_clock` is an alias for either `system_clock` or `steady_clock`, which means that the clock is sometimes steady and sometimes not, and that this clock might or might not support conversions to other clocks or `time_t`.

Because the `high_resolution_clock` is no finer than the `steady_clock` on any platform, you should use the `steady_clock` instead.[2]

We will discuss the following details of clocks later:

- The difference between the clocks in detail
- Conversions between clocks
- How clocks deal with leap seconds

11.3.3 Timepoint Types

C++20 introduces a couple of new types for timepoints. Based on the general definition available since C++11:

```
template<typename Clock, typename Duration = typename Clock::duration>
class time_point;
```

the new types provide a more convenient way of using timepoints with the different clocks.

Table *Standard timepoint types since C++20* describes the names and meanings of the convenient time-point types.

Type	Meaning	Defined as
`local_time`<*Dur*>	Local timepoint	`time_point<`*LocalTime*`, `*Dur*`>`
`local_seconds`	Local timepoint in seconds	`time_point<`*LocalTime*`, seconds>`
`local_days`	Local timepoint in days	`time_point<`*LocalTime*`, days>`
`sys_time`<*Dur*>	System timepoint	`time_point<system_clock, `*Dur*`>`
`sys_seconds`	System timepoint in seconds	`time_point<system_clock, seconds>`
`sys_days`	System timepoint in days	`time_point<system_clock, days>`
`utc_time`<*Dur*>	UTC timepoint	`time_point<utc_clock, `*Dur*`>`
`utc_seconds`	UTC timepoint in seconds	`time_point<utc_clock, seconds>`
`tai_time`<*Dur*>	TAI timepoint	`time_point<tai_clock, `*Dur*`>`
`tai_seconds`	TAI timepoint in seconds	`time_point<tai_clock, seconds>`
`gps_time`<*Dur*>	GPS timepoint	`time_point<gps_clock, `*Dur*`>`
`gps_seconds`	GPS timepoint in seconds	`time_point<gps_clock, seconds>`
`file_time`<*Dur*>	Filesystem timepoint	`time_point<file_clock, `*Dur*`>`

Table 11.3. Standard timepoint types since C++20

For each standard clock (except the steady clock), we have a corresponding `_time<>` type, which allows us to declare objects that represent a date or timepoint of the clock. In addition, the ..._`seconds` types allow us to define date/time objects of the corresponding type with the granularity of seconds. For system and local time, the ..._`days` type allow us to define date/time objects of the corresponding type with the granularity of days. For example:

```
std::chrono::sys_days x;                     // time_point<system_clock, days>
```

[2] Thanks to Howard Hinnant for pointing this out.

```
std::chrono::local_seconds y;                    //time_point<local_t, seconds>
std::chrono::file_time<std::chrono::seconds> z; //time_point<file_clock, seconds>
```

Please note that objects of this type still represent one timepoint, although the type unfortunately has a plural name. For example, `sys_days` means a single day defined as a system timepoint (the name comes from "system timepoint with granularity days").

11.3.4 Calendrical Types

As an extension to timepoint types, C++20 introduces types for the civil (Gregorian) calendar to the chrono library.

While timepoints are specified as durations from an epoch, calendrical types have distinct and combined types and values for years, months, weekdays, and days of a month. Both are useful to use:

- Timepoint types are great as long as we only compute with seconds, hours, and days (such as doing something on each day of a year).
- Calendrical types are great for date arithmetic where the fact that months and years have a different number of days comes into account. In addition, you can deal with something like "the third Monday of the month" or "the last day of the month."

Table *Standard calendrical types* lists the new calendrical types together with their default output format.

Type	Meaning	Output Format
day	Day	05
month	Month	Feb
year	Year	1999
weekday	Weekday	Mon
weekday_indexed	*n*th weekday	Mon[2]
weekday_last	Last weekday	Mon[last]
month_day	Day of a month	Feb/05
month_day_last	Last day of a month	Feb/last
month_weekday	*n*th weekday of a month	Feb/Mon[2]
month_weekday_last	Last weekday of a month	Feb/Mon[last]
year_month	Month of a year	1999/Feb
year_month_day	A full date (day of a month of a year)	1999-02-05
year_month_day_last	Last day of a month of a year	1999/Feb/last
year_month_weekday	*n*th weekday of a month of a year	1999/Feb/Mon[2]
year_month_weekday_last	Last weekday of a month of a year	1999/Feb/Mon[last]

Table 11.4. Standard calendrical types

Remember that these type names do not imply the order in which the elements of the date can be passed for initialization or are written to formatted output. For example, type `std::chrono::year_month_day` can be used as follows:

```
using namespace std::chrono;

year_month_day d = January/31/2021;          // January 31, 2021
std::cout << std::format("{:%D}", d);        // writes 01/31/21
```

Calendrical types such as `year_month_day` allow us to compute precisely the date of the same day next month:

```
std::chrono::year_month_day start = ... ;           // 2021/2/5
auto end = start + std::chrono::months{1};          // 2021/3/5
```

If you use timepoints such as `sys_days`, the corresponding code would not really work because it uses the ***average*** duration of a month:

```
std::chrono::sys_days start = ... ;                 // 2021/2/5
auto end = start + std::chrono::months{1};          // 2021/3/7 10:29:06
```

Note that in this case, `end` has a different type because with `months`, fractional days come into play.

When adding 4 weeks or 28 days, timepoints types are better, because for them, this is a simple arithmetic operation and does not have to take the different lengths of months or years into account:

```
std::chrono::sys_days start = ... ;                 // 2021/1/5
auto end = start + std::chrono::weeks{4};           // 2021/2/2
```

Details of using `months` and `years` are discussed later.

As you can see, there are specific types for dealing with weekdays and the *n*th and last weekday within a month. This allows us to iterate over all second Mondays or jump to the last day of the next month:

```
std::chrono::year_month_day_last start = ... ;      // 2021/2/28
auto end = start + std::chrono::months{1};          // 2021/3/31
```

Every type has a default output format, which is a fixed sequence of English characters. For other formats, use the formatted chrono output. Please note that only `year_month_day` uses dashes as a separator in its default output format. All other types separate their tokens with a slash by default.

To deal with the values of the calendrical types, a couple of calendrical constants are defined:

- `std::chrono::`**`last`**
  to specify the last day/weekday of a month. The constant has type `std::chrono::`**`last_spec`**.
- `std::chrono::`**`Sunday`**, `std::chrono::`**`Monday`**, ... `std::chrono::`**`Saturday`**
  to specify a weekday (Sunday has value 0, Saturday has value 6).
  These constants have type `std::chrono::`**`weekday`**.
- `std::chrono::`**`January`**, `std::chrono::`**`February`**, ... `std::chrono::`**`December`**
  to specify a month (January has value 1, December has value 12).
  These constants have type `std::chrono::`**`month`**.

Restricted Operations for Calendrical Types

The calendrical types were designed to detect at compile-time that operations are not useful or do not have the best performance. As a result, some “obvious” operations do not compile as listed in table *Standard operations for calendrical types*:

- Day and month arithmetic depends on the year. You cannot add days/months or compute the difference of all month types that do not have a year.
- Day arithmetic for fully specified dates takes a bit of time to deal with the different lengths of months and leap years. You can add/subtract days or compute the difference between these types.
- Because chrono makes no assumption about the first day of a week, you cannot get an order between weekdays. When comparing types with weekdays, only `operator==` and `operator!=` are supported.

Type	++/−−	add/subtract	− (diff.)	==	</<=>
`day`	yes	`days`	yes	yes	yes
`month`	yes	`months`, `years`	yes	yes	yes
`year`	yes	`years`	yes	yes	yes
`weekday`	yes	`days`	yes	yes	-
`weekday_indexed`	-	-	-	yes	-
`weekday_last`	-	-	-	yes	-
`month_day`	-	-	-	yes	yes
`month_day_last`	-	-	-	yes	yes
`month_weekday`	-	-	-	yes	-
`month_weekday_last`	-	-	-	yes	-
`year_month`	-	`months`, `years`	yes	yes	yes
`year_month_day`	-	`months`, `years`	-	yes	yes
`year_month_day_last`	-	`months`, `years`	-	yes	yes
`year_month_weekday`	-	`months`, `years`	-	yes	-
`year_month_weekday_last`	-	`months`, `years`	-	yes	-

Table 11.5. Standard operations for calendrical types

Weekday arithmetic is modulo 7, which means that it does not really matter which day is the first day of a week. You can compute the difference between any two weekdays and the result is always a value between 0 and 6. Adding the difference to the first weekday will always be the second weekday. For example:

```
std::cout << chr::Friday - chr::Tuesday << '\n';          // 3d (Tuesday thru Friday)
std::cout << chr::Tuesday - chr::Friday << '\n';          // 4d (Friday thru Tuesday)

auto d1 = chr::February / 25 / 2021;
auto d2 = chr::March / 3 / 2021;
std::cout << chr::sys_days{d1} - chr::sys_days{d2} << '\n';   // -6d  (date diff)
std::cout << chr::weekday(d1) - chr::weekday(d2) << '\n';     // 3d   (weekday diff)
```

That way, you can always easily compute something like the difference to “the next Monday” as “Monday minus the current weekday:”

```
d1 =  chr::sys_days{d1} + (chr::Monday - chr::weekday(d1));   // set d1 to next Monday
```

Note that if `d1` is a calendrical date type, you first have to convert it to type `std::chrono::sys_days` so that day arithmetic is supported (it might be better to declare `d1` with this type).

In the same way, month arithmetic is modulo 12, which means that the next month after December is January. If a year is part of the type, it is adjusted accordingly:

```
auto m = chr::December;
std::cout << m + chr::months{10} << '\n';        //Oct
std::cout << 2021y/m + chr::months{10} << '\n';  //2022/Oct
```

Note also that the constructors of chrono calendrical types that take an integral value are `explicit`, meaning that explicit initialization with an integral value fails:

```
std::chrono::day d1{3};        // OK
std::chrono::day d2 = 3;       // ERROR

d1 = 3;                        // ERROR
d1 = std::chrono::day{3};      // OK

passDay(3);                    // ERROR
passDay(std::chrono::day{3});  // OK
```

11.3.5 Time Type hh_mm_ss

In accordance with the calendrical types, C++20 introduces a new time type `std::chrono::`**`hh_mm_ss`** that converts a duration to a data structure with corresponding time fields. Table `std::chrono::hh_mm_ss` *members* describes the names and meanings of the `hh_mm_ss` members.

Member	Meaning
`hours()`	Hours value
`minutes()`	Minutes value
`seconds()`	Seconds value
`subseconds()`	Value for partial seconds with appropriate granularity
`is_negative()`	True if the value is negative
`to_duration()`	Conversion (back) to duration
`precision`	Duration type of the subseconds
`operator precision()`	Conversion to value with corresponding precision
`fractional_width`	Precision of the subseconds (static member)

Table 11.6. `std::chrono::hh_mm_ss` *members*

This type is very useful for dealing with the different attributes of durations and timepoints. It allows us to split durations into its attributes and serves as a formatting aid. For example, you can check for a specific hour or pass the hour and minute to another function as integral values:

```
auto dur = measure();                   // process and yield some duration
std::chrono::hh_mm_ss hms{dur};         // convert to data structure for attributes
process(hms.hours(), hms.minutes());    // pass hours and minutes
```

If you have a timepoint, you have to convert it to a duration first. To do this, you usually just compute the difference between the timepoint and midnight of the day (computed by rounding it down to the granularity of days). For example:

```
auto tp = getStartTime();                  // process and yield some timepoint
// convert time to data structure for attributes:
std::chrono::hh_mm_ss hms{tp - std::chrono::floor<std::chrono:days>(tp)};
process(hms.hours(), hms.minutes());    // pass hours and minutes
```

As another example, we can use `hh_mm_ss` to print the attributes of a duration in a different form:

```
auto t0 = std::chrono::system_clock::now();
...
auto t1 = std::chrono::system_clock::now();
std::chrono::hh_mm_ss hms{t1 - t0};
std::cout << "minutes:  " << hms.hours() + hms.minutes() << '\n';
std::cout << "seconds:  " << hms.seconds() << '\n';
std::cout << "subsecs:  " << hms.subseconds() << '\n';
```

might print:

```
minutes:  63min
seconds:  19s
subsecs:  502998000ns
```

There is no way to directly initialize the different attributes of `hh_mm_ss` objects with specific values. In general, you should use duration types to deal with times:

```
using namespace std::literals;
...
auto t1 = 18h + 30min;      // 18 hours and 30 minutes
```

The most powerful functionality of `hh_mm_ss` is that it takes *any* precision duration and transforms it into the usual attributes and duration types `hours`, `minutes`, and `seconds`. In addition, `subseconds()` yields the rest of the value with an appropriate duration type, such as `milliseconds` or `nanoseconds`. Even if the unit is not a power of 10 (e.g., thirds of a second), `hh_mm_ss` transforms that into subseconds with a power of 10 with up to 18 digits. If that does not represent the exact value, a precision of six digits is used. You can use the standard output operator to print the result as a whole. For example:

```
std::chrono::duration<int, std::ratio<1,3>> third{1};
auto manysecs = 10000s;
auto dblsecs = 10000.0s;

std::cout << "third:    " << third << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{third} << '\n';
std::cout << "manysecs: " << manysecs << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{manysecs} << '\n';
std::cout << "dblsecs:  " << dblsecs << '\n';
std::cout << "          " << std::chrono::hh_mm_ss{dblsecs} << '\n';
```

This code has the following output:

```
third:    1[1/3]s
          00:00:00.333333
manysecs: 10000s
          02:46:40
dblsecs:  10000.000000s
          02:46:40.000000
```

To output specific attributes, you can also use formatted output with specific conversion specifiers:

```
auto manysecs = 10000s;
std::cout << "manysecs: " << std::format("{:%T}", manysecs) << '\n';
```

This also writes:

```
manysecs: 02:46:40
```

11.3.6 Hours Utilities

The chrono library now also provides a few helper functions to deal with the 12-hour and 24-hour formats. Table *Hours Utilities* lists these functions.

`std::chrono::`**`is_am`**`(`*h*`)`	Yields whether *h* is an `hours` value between 0 and 11
`std::chrono::`**`is_pm`**`(`*h*`)`	Yields whether *h* is an `hours` value between 12 and 23
`std::chrono::`**`make12`**`(`*h*`)`	Yields the 12-hour equivalent of the `hours` value *h*
`std::chrono::`**`make24`**`(`*h*`, `*toPM*`)`	Yields the 24-hour equivalent of the `hours` value *h*

Table 11.7. Hours utilities

Both `std::chrono::make12()` and `std::chrono::make24()` require an hours value between 0 and 23. The second parameter of `std::chrono::make24()` specifies whether to interpret the passed hours value as PM value. If `true`, the function adds 12 hours to the passed value, if it is between 0 and 11.

For example:

```
for (int hourValue : {9, 17}) {
  std::chrono::hours h{hourValue};
  if (std::chrono::is_am(h)) {
    h = std::chrono::make24(h, true);          // assume a PM hour is meant
  }
  std::cout << "Tea at " << std::chrono::make12(h).count() << "pm" << '\n';
}
```

The code has the following output:

```
Tea at 9pm
Tea at 5pm
```

11.4 I/O with Chrono Types

C++20 provides new support for directly outputting and parsing almost all chrono types.

11.4.1 Default Output Formats

For more or less all chrono types, the standard output operator has been defined since C++20. If useful, it not only prints the value but also uses an appropriate format and the unit. Locale-dependent formatting is possible.

All **calendrical types** print out the value as listed with the output formats of calendrical types.

All **duration types** print out the value with the unit type as listed in table *Output units for durations*. This fits any literal operator if provided.

Unit	**Output Suffix**
`atto`	`as`
`femto`	`fs`
`pico`	`ps`
`nano`	`ns`
`micro`	`µs` or `us`
`milli`	`ms`
`centi`	`cs`
`deci`	`ds`
`ratio<1>`	`s`
`deca`	`das`
`hecto`	`hs`
`kilo`	`ks`
`mega`	`Ms`
`giga`	`Gs`
`tera`	`Ts`
`peta`	`Ps`
`exa`	`Es`
`ratio<60>`	`min`
`ratio<3600>`	`h`
`ratio<86400>`	`d`
`ratio<`*num*`,1>`	`[`*num*`]s`
`ratio<`*num*`,`*den*`>`	`[`*num*`/`*den*`]s`

Table 11.8. Output units for durations

For all standard **timepoint** types, an output operator prints date and optionally time in the following format:

- *year*−*month*−*day* for an integral granularity unit implicitly convertible to days
- *year*−*month*−*day hour*:*minutes*:*seconds* for an integral granularity unit equal to or less than days

If the type of the timepoint value (member rep) has a floating-point type, no output operator is defined.[3]

For the time part, the output operator of type hh_mm_ss is used. This corresponds with the %F %T conversion specifier for formatted output.

For example:

```
auto tpSys = std::chrono::system_clock::now();
std::cout << tpSys << '\n';                      //2021-04-25 13:37:02.936314000

auto tpL = chr::zoned_time{chr::current_zone(), tpSys}.get_local_time();
std::cout << tpL;                                //2021-04-25 15:37:02.936314000
std::cout << chr::floor<chr::milliseconds>(tpL); //2021-04-25 15:37:02.936
std::cout << chr::floor<chr::seconds>(tpL);      //2021-04-25 15:37:02
std::cout << chr::floor<chr::minutes>(tpL);      //2021-04-25 15:37:00
std::cout << chr::floor<chr::days>(tpL);         //2021-04-25
std::cout << chr::floor<chr::weeks>(tpL);        //2021-04-22

auto tp3 = std::chrono::floor<chr::duration<long long, std::ratio<1, 3>>>(tpSys);
std::cout << tp3 << '\n';                        //2021-04-25 13:37:02.666666

chr::sys_time<chr::duration<double, std::milli>> tpD{tpSys};
std::cout << tpD << '\n';                        // ERROR: no output operator defined

std::chrono::gps_seconds tpGPS;
std::cout << tpGPS << '\n';                      //1980-01-06 00:00:00

auto tpStd = std::chrono::steady_clock::now();
std::cout << "tpStd: " << tpStd;                 // ERROR: no output operator defined
```

The zoned_time<> type outputs like the timepoint type extended with the abbreviated timezone name. The following timezone abbreviations are used for the standard clocks:

- UTC for sys_clock, utc_clock, and file_clock
- TAI for tai_clock
- GPS for gps_clock

Finally, note that sys_info and local_info have output operators with an undefined format. They should only be used for debugging purposes.

11.4.2 Formatted Output

Chrono supports the new library for formatted output. That means you can use date/time types for arguments of std::format() and std::format_to().

For example:

```
auto t0 = std::chrono::system_clock::now();
```

[3] This is a gap that will hopefully be fixed with C++23.

```
...
auto t1 = std::chrono::system_clock::now();

std::cout << std::format("From {} to {}\nit took {}\n", t0, t1, t1-t0);
```

This would use the default output formats of the date/time types. For example, we might have the following output:

```
From 2021-04-03 15:21:33.197859000 to 2021-04-03 15:21:34.686544000
it took 1488685000ns
```

To improve the output, you can constrain the types or use specific conversion specifiers. Most of them (but not all) correspond with the C function `strftime()` and the POSIX `date` command. For example:

```
std::cout << std::format("From {:%T} to {:%T} it took {:%S}s\n", t0, t1, t1-t0);
```

This might have the following output then:

```
From 15:21:34.686544000 to 15:21:34.686544000 it took 01.488685000s
```

The format specifiers for chrono types are part of the syntax for standard format specifiers (each specifier is optional):

fill align width `.`*prec* **`L`** *spec*

- *fill*, *align*, *with*, and *prec* mean the same as for standard format specifiers.
- *spec* specifies the general notation of the formatting, starting with `%`.
- **`L`** also as usual turns on locale-dependent formatting for specifiers that support it.

Table *Conversion specifiers for chrono types* lists all conversion specifiers for formatted output of date/time types with an example based on Sunday, June 9, 2019 17:33:16 and 850 milliseconds.

Without using a specific conversion specifier, the default output operator is used, which marks invalid dates. When using specific conversion specifiers, this is not the case:

```
std::chrono::year_month_day ymd{2021y/2/31};    // February 31, 2021
std::cout << std::format("{}", ymd);            // 2021-02-31 is not a valid ...
std::cout << std::format("{:%F}", ymd);         // 2021-02-31
std::cout << std::format("{:%Y-%m-%d}", ymd);   // 2021-02-31
```

If the date/time value type does not provide the necessary information for a conversion specifier, a `std::format_error` exception is thrown. For example, an exception is thrown when:

- A year specifier is used for a `month_day`
- A weekday specifier is used for a duration
- A month or weekday name should be printed and the value is not valid
- A timezone specifier is used for a local timepoint

In addition, note the following about conversion specifiers:

- A negative duration or `hh_mm_ss` value print the minus sign in front of the whole value. For example:

```
    std::cout << std::format("{:%H:%M:%S}", -10000s);   // outputs: -02:46:40
```

Spec.	Example	Meaning
`%c`	`Sun Jun  9 17:33:16 2019`	Standard or locale's date and time representation
Dates:		
`%x`	`06/09/19`	Standard or locale's date representation
`%F`	`2019-06-09`	*year*−*month*−*day* with four and two digits
`%D`	`06/09/19`	*month*/*day*/*year* with two digits
`%e`	`9`	Day with leading space if single digit
`%d`	`09`	Day as two digits
`%b`	`Jun`	Standard or locale's abbreviated month name
`%h`	`Jun`	ditto
`%B`	`June`	Standard or locale's month name
`%m`	`06`	Month with two digits
`%Y`	`2019`	Year with four digits
`%y`	`19`	Year without century as two digits
`%G`	`2019`	ISO-week-based year as four digits (week according to `%V`)
`%g`	`19`	ISO-week-based year as two digits (week according to `%V`)
`%C`	`20`	Century as two digits
Weekdays and weeks:		
`%a`	`Sun`	Standard or locale's abbreviated weekday name
`%A`	`Sunday`	Standard or locale's weekday name
`%w`	`0`	Weekday as decimal number (Sunday as `0` until Saturday as `6`)
`%u`	`7`	Weekday as decimal number (Monday as `1` until Sunday as `7`)
`%W`	`22`	Week of the year (`00` ... `53`, week `01` starts with first Monday)
`%U`	`23`	Week of the year (`00` ... `53`, week `01` starts with first Sunday)
`%V`	`23`	ISO week of the year (`01` ... `53`, week `01` has Jan. 4th)
Times:		
`%X`	`17:33:16`	Standard or locale's time representation
`%r`	`05:33:16 PM`	Standard or locale's 12-hour clock time
`%T`	`17:33:16.850`	*hour*:*minutes*:*seconds* (locale-dependent subseconds as needed)
`%R`	`17:33`	*hour*:*minutes* with two digits each
`%H`	`17`	24-hour clock hour as two digits
`%I`	`05`	12-hour clock hour as two digits
`%p`	`PM`	`AM` or `PM` according to the 12-hour clock
`%M`	`33`	Minute with two digits
`%S`	`16.850`	Seconds as decimal number (locale-specific subseconds as needed)
Other:		
`%Z`	`CEST`	Timezone abbreviation (may also be `UTC`, `TAI`, or `GPS`)
`%z`	`+0200`	Offset (hours and minutes) from UTC (`%Ez` or `%Oz` for `+02:00`)
`%j`	`160`	Day of the year with three digits (Jan. 1st is `001`)
`%q`	`ms`	Unit suffix according to the time's duration
`%Q`	`63196850`	Value according to the time's duration
`%n`	`\n`	Newline character
`%t`	`\t`	Tabulator character
`%%`	`%`	`%` character

Table 11.9. Conversion specifiers for chrono types

- Different week number and year formats might result in different output values.
  For example, Sunday, January 1, 2023 yields:
  - Week 00 with %W (week before first Monday)
  - Week 01 with %U (week with first Monday)
  - Week 52 with %V (ISO week: week before Monday of week 01, which has January 4th)

  Because the ISO week might be the last week of the previous year, the ISO year, which is the year of that week, may be one less:
  - Year 2023 with %Y
  - Year 23 with %y
  - Year 2022 with %G (ISO year of the ISO week %V, which is the last week of the previous month)
  - Year 22 with %g (ISO year of the ISO week %V, which is the last week of the previous month)
- The following timezone abbreviations are used for the standard clocks:
  - UTC for sys_clock, utc_clock, and file_clock
  - TAI for tai_clock
  - GPS for gps_clock

All conversion specifiers except %q and %Q can also be used for formatted parsing.

11.4.3 Locale-Dependent Output

The default output operator for the various types uses a locale-dependent format if the output stream is imbued by a locale that has its own format. For example:

```
using namespace std::literals;
auto dur = 42.2ms;

std::cout << dur << '\n';     // 42.2ms

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif
std::cout.imbue(locG);        // switch to German locale
std::cout << dur << '\n';     // 42,2ms
```

Formatted output with std::format() is handled as usual:[4]

- By default, formatted output uses the locale-independent "C" locale.
- By specifying L, you can switch to a locale-dependent output specified via a locale parameter or as a global locale.

[4] This behavior was specified as a bug fix to C++20 with http://wg21.link/p2372, which means that the original wording of C++20 does not specify this behavior.

This means that to use a locale-dependent notation, you have to use the `L` specifier and either pass the locale as the first argument to `std::format()` or set the global locale before calling it. For example:

```
using namespace std::literals;
auto dur = 42.2ms;                              // duration to print

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif

std::string s1 = std::format("{:%S}", dur);        // "00.042s" (not localized)
std::string s3 = std::format(locG, "{:%S}", dur);   // "00.042s" (not localized)
std::string s2 = std::format(locG, "{:L%S}", dur);  // "00,042s" (localized)

std::locale::global(locG);                      // set German locale globally
std::string s4 = std::format("{:L%S}", dur);       // "00,042s" (localized)
```

In several cases, you can even use an alternative locale's representation according to `strftime()` and ISO 8601:2004, which you can specify with a leading `O` or `E` in front of the conversion specifier:

- `E` can be used as the locale's alternate representations in front of `c`, `C`, `x`, `X`, `y`, `Y`, and `z`
- `O` can be used as the locale's alternate numeric symbols in front of `d`, `e`, `H`, `I`, `m`, `M`, `S`, `u`, `U` `V`, `w`, `W`, `y`, and `z`

11.4.4 Formatted Input

The chrono library also supports formatted input. You have two options:

- A free-standing function `std::chrono::`**`from_stream()`** is provided by certain date/time types to read in a specific value according to a passed format string.
- A manipulator `std::chrono::`**`parse()`** allows us to use `from_stream()` as part of a bigger parsing with the input operator >>.

Using `from_stream()`

The following code demonstrates how to use `from_stream()` by parsing a full timepoint:

```
std::chrono::sys_seconds tp;
std::istringstream sstrm{"2021-2-28 17:30:00"};

std::chrono::from_stream(sstrm, "%F %T", tp);
if (sstrm) {
  std::cout << "tp: " << tp << '\n';
}
else {
  std::cerr << "reading into tp failed\n";
}
```

The code generates the following output:

```
tp: 2021-02-28 17:30:00
```

As another example, you can parse a `year_month` from a sequence of characters specifying the full month name and the year, among other things, as follows:

```
std::chrono::year_month m;
std::istringstream sstrm{"Monday, April 5, 2021"};
std::chrono::from_stream(sstrm, "%A, %B %d, %Y", m);
if (sstrm) {
  std::cout << "month: " << m << '\n';  // prints: month: 2021/Apr
}
```

The format string accepts all conversion specifiers of formatted output except `%q` and `%Q` with improved flexibility. For example:

- `%d` stands for one or two characters to specify the day and with `%4d`, you can specify that even up to four characters are parsed.
- `%n` stands for exactly one whitespace character.
- `%t` stands for zero or one whitespace character.
- A whitespace character such as a space represents an arbitrary number of whitespaces (including zero whitespaces).

`from_stream()` is provided for the following types:

- A `duration<>` of any type
- A `sys_time<>`, `utc_time<>`, `gps_time<>`, `tai_time<>`, `local_time<>`, or `file_time<>` of any duration
- A `day`, a `month`, or a `year`
- A `year_month`, a `month_day`, or a `year_month_day`
- A `weekday`

The format has to be a C string of type `const char*`. It must match the characters in the input stream and the value to parse. The parsing fails if:

- The input sequence of characters does not match the required format
- The format does not provide enough information for the value
- The parsed date is not valid

In that case, the failbit of the stream is set, which you can test with by calling `fail()` or using the stream as a Boolean value.

A Generic Function for Parsing Dates/Times

In practice, dates and times are rarely hard-coded. However, when testing code, you often need an easy way to specify a date/time value.

Here is a little helper function I used to test the examples in this book:

lib/chronoparse.hpp

```
#include <chrono>
#include <string>
#include <sstream>
#include <cassert>

// parse year-month-day with optional hour:minute and optional :sec
// - returns a time_point<> of the passed clock (default: system_clock)
//   in seconds
template<typename Clock = std::chrono::system_clock>
auto parseDateTime(const std::string& s)
{
  // return value:
  std::chrono::time_point<Clock, std::chrono::seconds> tp;

  // string stream to read from:
  std::istringstream sstrm{s};   // no string_view support

  auto posColon = s.find(":");
  if (posColon != std::string::npos) {
    if (posColon != s.rfind(":")) {
      // multiple colons:
      std::chrono::from_stream(sstrm, "%F %T", tp);
    }
    else {
      // one colon:
      std::chrono::from_stream(sstrm, "%F %R", tp);
    }
  }
  else {
    // no colon:
    std::chrono::from_stream(sstrm, "%F", tp);
  }

  // handle invalid formats:
  assert((!sstrm.fail()));

  return tp;
}
```

You can use `parseDateTime()` as follows:

lib/chronoparse.cpp

```
#include "chronoparse.hpp"
#include <iostream>

int main()
{
  auto tp1 = parseDateTime("2021-1-1");
  std::cout << std::format("{:%F %T %Z}\n", tp1);

  auto tp2 = parseDateTime<std::chrono::local_t>("2021-1-1");
  std::cout << std::format("{:%F %T}\n", tp2);

  auto tp3 = parseDateTime<std::chrono::utc_clock>("2015-6-30 23:59:60");
  std::cout << std::format("{:%F %T %Z}\n", tp3);

  auto tp4 = parseDateTime<std::chrono::gps_clock>("2021-1-1 18:30");
  std::cout << std::format("{:%F %T %Z}\n", tp4);
}
```

The program has the following output:

```
2021-01-01 00:00:00 UTC
2021-01-01 00:00:00
2015-06-30 23:59:60 UTC
2021-01-01 18:30:00 GPS
```

Note that for a local timepoint, you cannot use `%Z` to print its timezone (doing so would raise an exception).

Using the `parse()` Manipulator

Instead of calling `from_stream()`

```
std::chrono::from_stream(sstrm, "%F %T", tp);
```

you could also call:

```
sstrm >> std::chrono::parse("%F %T", tp);
```

Please note that the original C++20 standard does not formally allow you to pass the format directly as a string literal, which means that you have to call

```
sstrm >> std::chrono::parse(std::string{"%F %T"}, tp);
```

However, this should be fixed with http://wg21.link/lwg3554.

`std::chrono::parse()` is an I/O stream manipulator. It allows you to parse multiple values inside one statement reading from an input stream. In addition, thanks to move semantics, you can even pass a temporary input stream. For example:

```
chr::sys_days tp;
chr::hours h;
chr::minutes m;
// parse date into tp, hour into h and minute into m:
std::istringstream{"12/24/21 18:00"} >> chr::parse("%D", tp)
                                     >> chr::parse(" %H", h)
                                     >> chr::parse(":%M", m);
std::cout << tp << " at " << h << ' ' << m << '\n';
```

This code outputs:

```
2021-12-24 at 18h 0min
```

Again, note that you might have to explicitly convert the string literals `"%D"`, `" %H"`, and `":%M"` to strings.

Parsing Timezones

Parsing timezones is a little tricky because timezone abbreviations are not unique:

To help here, `from_stream()` has the following formats:

istream **`from_stream(`***istream*, *format*, *value***`)`**
istream **`from_stream(`***istream*, *format*, *value*, *abbrevPtr***`)`**
istream **`from_stream(`***istream*, *format*, *value*, *abbrevPtr*, *offsetPtr***`)`**

As you can see, you can optionally pass the address of a `std::string` to store a parsed timezone abbreviation into the string and the address of a `std::chrono::minutes` object to store a parsed timezone offset into that string. In both cases `nullptr` can be passed.

However, you still have to be careful:

- The following works:

  ```
  chr::sys_seconds tp;
  std::istringstream sstrm{"2021-4-13 12:00 UTC"};
  chr::from_stream(sstrm, "%F %R %Z", tp);
  std::cout << std::format("{:%F %R %Z}\n", tp);  // 2021-04-13 12:00 UTC
  ```

  However, it works only because system timepoints use `UTC` anyway.
- The following does not work because it ignores the timezone:

  ```
  chr::sys_seconds tp;
  std::istringstream sstrm{"2021-4-13 12:00 MST"};
  chr::from_stream(sstrm, "%F %R %Z", tp);
  std::cout << std::format("{:%F %R %Z}", tp);  // 2021-04-13 12:00 UTC
  ```

  `%Z` is used to parse `MST` but there is no parameter to store the value.
- The following seems to work:

  ```
  chr::sys_seconds tp;
  std::string tzAbbrev;
  std::istringstream sstrm{"2021-4-13 12:00 MST"};
  chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
  ```

```
std::cout << tp << '\n';        //2021-04-13 12:00
std::cout << tzAbbrev << '\n'; //MST
```

However, if you compute the zoned time, you see that you are converting a UTC time to a different timezone:

```
chr::zoned_time zt{tzAbbrev, tp};      // OK: MST exists
std::cout << zt << '\n';        //2021-04-13 05:00:00 MST
```

- The following really does seem to work:

```
chr::local_seconds tp;   // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << tp << '\n';        //2021-04-13 12:00
std::cout << tzAbbrev << '\n'; //MST
chr::zoned_time zt{tzAbbrev, tp};      // OK: MST exists
std::cout << zt << '\n';        //2021-04-13 12:00:00 MST
```

  However, we were lucky that `MST` is one of the few abbreviations available as a deprecated entry in the timezone database. The moment you use this code with `CEST` or `CST`, it throws an exception when initializing the `zoned_time`.

- Therefore, either use only `tzAbbrev` instead of `zoned_time` and `%Z`:

```
chr::local_seconds tp;   // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 CST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << std::format("{:%F %R} {}", tp, tzAbbrev);  //2021-04-13 12:00 CST
```

  or you have to deal with code to map a timezone abbreviation to a timezone.

Note that `%Z` cannot parse the pseudo timezones `GPS` and `TAI`.

11.5 Using the Chrono Extensions in Practice

Now that you have learned about the new features and types of the chrono library, this section discusses how to use them in practice.

See also http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes for more examples and recipes.

11.5.1 Invalid Dates

Values of calendrical types may not be valid. This can happen in two ways:

- By an initialization with an invalid value. For example:

```
std::chrono::day d{0};                        // invalid day
std::chrono::year_month ym{2021y/13};         // invalid year_month
std::chrono::year_month_day ymd{2021y/2/31};  // invalid year_month_day
```

- By a computation that results in an invalid date. For example:

```
auto ymd1 = std::chrono::year{2021}/1/31;     // January 31, 2021
ymd1 += std::chrono::months{1};               // February 31, 2021  (invalid)

auto ymd0 = std::chrono::year{2020}/2/29;     // February 29, 2020
ymd1 += std::chrono::years{1};                // February 29, 2021  (invalid)
```

Table *Valid values of standard date attributes* lists the internal types and possible values of the different attributes of a date.

Attribute	Internal Type	Valid Values
Day	`unsigned char`	1 to 31
Month	`unsigned char`	1 to 12
Year	`short`	-32767 to 32767
Weekday	`unsigned char`	0 (Sunday) to 6 (Saturday)
		and 7 (again Sunday), which is converted to 0
Weekday index	`unsigned char`	1 to 5

Table 11.10. Valid values of standard date attributes

All combined types are valid if the individual components are valid (e.g., a valid `month_weekday` is valid if both the month and the weekday are valid). However, additional checks may apply:

- A full date of type `year_month_day` has to exist which means that it takes leap years into account. For example:

```
2020y/2/29;           // valid (there is a February 29 in 2020)
2021y/2/29;           // invalid (there is no February 29 in 2021)
```

- A `month_day` is valid only if the day could be valid in that month. For February, 29 is valid but 30 is not. For example:

```
February/29;          // valid (a February can have 29 days)
February/30;          // invalid (no February can have 30 days)
```

- A `year_month_weekday` is valid only if the weekday index can exist in the specified month of a year. For example:

  ```
  2020y/1/Thursday[5];   // valid (there is a fifth Thursday in January 2020)
  2020y/1/Sunday[5];     // invalid (there is no fifth Sunday in January 2020)
  ```

Each calendrical type provides a member function `ok()` to check whether the value is valid. Default output operators signal invalid dates.

The way to handle invalid dates depends on your programming logic. For the typical scenario where you have created a day too high for a month, you have the following options:

- Round down to the last day of the month:

  ```
  auto ymd = std::chrono::year{2021}/1/31;
  ymd += std::chrono::months{1};
  if (!ymd.ok()) {
    ymd = ymd.year()/ymd.month()/std::chrono::last;            // February 28, 2021
  }
  ```

  Note that the expression on the right creates a `year_month_last` which is then converted to type `year_month_day`.
- Round up to the first day of the next month:

  ```
  auto ymd = std::chrono::year{2021}/1/31;
  ymd += std::chrono::months{1};
  if (!ymd.ok()) {
    ymd = ymd.year()/ymd.month()/1 + std::chrono::months{1};   // March 1, 2021
  }
  ```

  Do not just add 1 to the month because for December, you create an invalid month.
- Round up to the date according to all overflown days:

  ```
  auto ymd = std::chrono::year{2021}/1/31;
  ymd += std::chrono::months{1};                               // March 3, 2021
  if (!ymd.ok()) {
    ymd = std::chrono::sys_days(ymd);
  }
  ```

  This uses a special feature of the conversion of a `year_month_day` where all overflown days are logically added to the next month. However, this does work only for a few days (you cannot add 1000 days that way).

If the date is not a valid value, the default output format will signal that with "is not a valid *type*." For example:

```
std::chrono::day d{0};                        // invalid day
std::chrono::year_month_day ymd{2021y/2/31};  // invalid year_month_day
std::cout << "day: " << d << '\n';
std::cout << "ymd: " << ymd << '\n';
```

This code will output:[5]

```
day: 00 is not a valid day
ymd: 2021-02-31 is not a valid year_month_day
```

The same happens when using the default formatting (just {}) of formatted output. By using specific conversion specifiers, you can disable the "*is not a valid*" output (in software for banking or quarterly processing, a day such as June 31 is sometimes even used):

```
std::chrono::year_month_day ymd{2021y/2/31};
std::cout << ymd << '\n';          // "2021-02-31 is not a valid year_month_day"
std::cout << std::format("{:%F}\n", ymd);          // "2021-02-31"
std::cout << std::format("{:%Y-%m-%d}\n", ymd);    // "2021-02-31"
```

11.5.2 Dealing with `months` and `years`

Because `std::chrono::months` and `std::chrono::years` are not integral multiples of days, you have to be careful when using them.

- For standard types that have their own value just for the year (`month`, `year`, `year_month`, `year_month_day`, `year_month_weekday_last`, ...), they work fine when adding a specific number of months or years to a date.
- For standard timepoint types that have one value for the date as a whole (`time_point`, `sys_time`, `sys_seconds`, and `sys_days`), they add the corresponding average fractional period to a date, which might not result in the date you expect.

For example, let us look at the different effects of adding four months or four years to December 31, 2020:

- When dealing with `year_month_day`:

  ```
  chr::year_month_day ymd0 = chr::year{2020}/12/31;
  auto ymd1 = ymd0 + chr::months{4};                // OOPS: April 31, 2021
  auto ymd2 = ymd0 + chr::years{4};                 // OK: December 31, 2024

  std::cout << "ymd:         " << ymd0 << '\n';    // 2020-12-31
  std::cout << " +4months:   " << ymd1 << '\n';    // 2021-04-31 is not a valid ...
  std::cout << " +4years:    " << ymd2 << '\n';    // 2024-12-31
  ```

- When dealing with `year_month_day_last`:

  ```
  chr::year_month_day_last yml0 = chr::year{2020}/12/chr::last;
  auto yml1 = yml0 + chr::months{4};                // OK: last day of April 2021
  auto yml2 = yml0 + chr::years{4};                 // OK: last day of Dec. 2024

  std::cout << "yml:         " << yml0 << '\n';    // 2020/Dec/last
  std::cout << " +4months:   " << yml1 << '\n';    // 2021/Apr/last
  ```

[5] Note that the specification in the C++20 standard is a little inconsistent here and is currently fixed. For example, it states that for an invalid `year_month_days` "*is not a valid **date***" is the output.

```
    std::cout << " as date:   "
              << chr::sys_days{yml1} << '\n';      // 2021-04-30
    std::cout << " +4years:   " << yml2 << '\n';   // 2024/Dec/last
```

- When dealing with sys_days:

```
    chr::sys_days day0 = chr::year{2020}/12/31;
    auto day1 = day0 + chr::months{4};              // OOPS: May 1, 2021 17:56:24
    auto day2 = day0 + chr::years{4};               // OOPS: Dec. 30, 2024 23:16:48

    std::cout << "day:         " << day0 << '\n';   // 2020-12-31
    std::cout << " with time: "
              << chr::sys_seconds{day0} << '\n';    // 2020-12-31 00:00:00
    std::cout << " +4months:  " << day1 << '\n';    // 2021-05-01 17:56:24
    std::cout << " +4years:   " << day2 << '\n';    // 2024-12-30 23:16:48
```

  This feature is supported only to allow you to model things such as physical or biological processes that do not care about the intricacies of human calendars (weather, gestation periods, etc.) and the feature should not be used for other purposes.

Note that both the output values and the default output formats differ. That is a clear sign that different types are used:

- When you add months or years to calendrical types, these types process the correct logical date (the same day or again the last day of the next month or year). Please note that this might result in invalid dates such as April 31, which the default output operator even signals in its output as follows:[6]

```
  2021-04-31 is not a valid year_month_day
```

- When you add months or years to a day of type `std::chrono::sys_days`, the result is not of type `sys_days`. As usual in the chrono library, the result has the best type that is able to represent any possible result:
  - Adding `months` yields a type with a unit of 54 seconds.
  - Adding `years` yields a type with a unit of 216 seconds.

  Both units are a multiple of a second, which means that they can be used as `std::chrono::sys_seconds`. By default, the corresponding output operator prints both the day and the time in seconds, which, as you can see, is not the same day and time of a later month or year. Both timepoints are no longer the 31st or last day of a month and the time is no longer midnight:

```
  2021-05-01 17:56:24
  2024-12-30 23:16:48
```

Both can be useful. However, using `months` and `years` with timepoints is usually only useful for computing the approximate day after many months and/or years.

[6] Due to some inconsistencies, the exact format of invalid dates might not match the C++20 standard, which specifies "`is not a valid date`" here.

11.5.3 Parsing Timepoints and Durations

If you have a timepoint or duration, you can access the different fields as demonstrated by the following program:

lib/chronoattr.cpp

```cpp
#include <chrono>
#include <iostream>

int main()
{
  auto now = std::chrono::system_clock::now();              // type is sys_time<>
  auto today = std::chrono::floor<std::chrono::days>(now);   // type is sys_days
  std::chrono::year_month_day ymd{today};
  std::chrono::hh_mm_ss hms{now - today};
  std::chrono::weekday wd{today};

  std::cout << "now:       " << now << '\n';
  std::cout << "today:     " << today << '\n';
  std::cout << "ymd:       " << ymd << '\n';
  std::cout << "hms:       " << hms << '\n';
  std::cout << "year:      " << ymd.year() << '\n';
  std::cout << "month:     " << ymd.month() << '\n';
  std::cout << "day:       " << ymd.day() << '\n';
  std::cout << "hours:     " << hms.hours() << '\n';
  std::cout << "minutes:   " << hms.minutes() << '\n';
  std::cout << "seconds:   " << hms.seconds() << '\n';
  std::cout << "subsecs:   " << hms.subseconds() << '\n';
  std::cout << "weekday:   " << wd << '\n';

  try {
    std::chrono::sys_info info{std::chrono::current_zone()->get_info(now)};
    std::cout << "timezone: " << info.abbrev << '\n';
  }
  catch (const std::exception& e) {
    std::cerr << "no timezone database: (" << e.what() << ")\n";
  }
}
```

The output of the program might be, for example, as follows:

```
now:       2021-04-02 13:37:34.059858000
today:     2021-04-02
ymd:       2021-04-02
hms:       13:37:34.059858000
year:      2021
```

```
month:    Apr
day:      02
hours:    13h
minutes:  37min
seconds:  34s
subsecs:  59858000ns
weekday:  Fri
timezone: CEST
```

The function `now()` yields a timepoint with the granularity of the system clock:

```
auto now = std::chrono::system_clock::now();
```

The result has type `std::chrono::sys_time<>` with some implementation-specific duration type for the resolution.

If you want to deal with the local time, you have to implement the following:

```
auto tpLoc = std::chrono::zoned_time{std::chrono::current_zone(),
                                     std::chrono::system_clock::now()
                                    }.get_local_time();
```

To deal with the date part of the timepoint, we need the granularity of days, which we get with:

```
auto today = std::chrono::floor<std::chrono::days>(now);
```

The initialized variable `today` has the type `std::chrono::sys_days`. You can assign this to an object of type `std::chrono::year_month_day`, so that you can access year, month, and day with the corresponding member functions:

```
std::chrono::year_month_day ymd{today};
std::cout << "year:     " << ymd.year() << '\n';
std::cout << "month:    " << ymd.month() << '\n';
std::cout << "day:      " << ymd.day() << '\n';
```

To deal with the time part of the timepoint we need a duration, which we get when we compute the difference between our original timepoint and its value at midnight. We use the duration to initialize a `hh_mm_ss` object:

```
std::chrono::hh_mm_ss hms{now - today};
```

From this, you can get the hour, minute, second, and subseconds directly:

```
std::cout << "hours:    " << hms.hours() << '\n';
std::cout << "minutes:  " << hms.minutes() << '\n';
std::cout << "seconds:  " << hms.seconds() << '\n';
std::cout << "subsecs:  " << hms.subseconds() << '\n';
```

For the subseconds, `hh_mm_ss` determines the necessary granularity and uses the appropriate unit. In our case, it is printed as nanoseconds:

```
hms:      13:37:34.059858000
hours:    13h
minutes:  37min
seconds:  34s
```

```
 subsecs:  59858000ns
```

For the weekday, you only have to initialize it with a type of granularity day. This works for `sys_days`, `local_days`, and `year_month_day` (because the latter implicitly converts to `std::sys_days`):

```
std::chrono::weekday wd{today};   // OK (today has day granularity)
std::chrono::weekday wd{ymd};     // OK due to implicit conversion to sys_days
```

For timezone aspects, you have to combine the timepoint (here `now`) with a timezone (here the current timezone). The resulting `std::chrono::sys_info` object contains information such as the abbreviated timezone name:

```
std::chrono::sys_info info{std::chrono::current_zone()->get_info(now)};
std::cout << "timezone: " << info.abbrev << '\n';
```

Note that not every C++ platform will support the timezone database. Therefore, on some systems this part of the program might throw an exception.

11.6 Timezones

In large countries or for international communication, it is not enough to say “let us meet at noon” because we have different timezones. The new chrono library supports this fact by having an API to deal with the different timezones we have on Earth, including dealing with *standard* (“winter”) and *daylight saving* (“summer”) time.

11.6.1 Characteristics of Timezones

Dealing with timezones is a little tricky because it is a topic that can become pretty complex. For example, you have to take the following into account:

- Timezone differences are not necessarily multiples of hours. In fact, we also have differences of 30 or even 15/45 minutes. For example, a significant part of Australia (the Northern Territory and South Australia with Adelaide) has the standard timezone of UTC+9:30 and Nepal has the timezone of UTC+5:45. Two timezones may also have 0 minutes difference (such as timezones in North and South America).
- Timezone abbreviations might refer to different timezones. For example, ***CST*** might stand for *Central Standard Time* (the standard timezone of Chicago, Mexico City, and Costa Rica) or *China Standard Time* (the international name of the timezone of Beijing and Shanghai), or *Cuba Standard Time* (the standard timezone of Havana).
- Timezones change. This can happen, for example, when countries decide to change their zone(s) or when daylight saving time starts. As a consequence, you might have to take multiple updates per year into account when dealing with timezones.

11.6.2 The IANA Timezone Database

To deal with timezones, the C++ standard library uses the IANA timezone database, which is available at `http://www.iana.org/time-zones`. Note again that the timezone database might not be available on all platforms.

The key entry for the timezone database is a timezone name, which is usually

- A ***city*** in a certain area or country representing the timezone.
  For example: `America/Chicago`, `Asia/Hong_Kong`, `Europe/Berlin`, `Pacific/Honolulu`
- A ***GMT entry*** with the negated offset from UTC time.
  For example: `Etc/GMT`, `Etc/GMT+6`, which stands for `UTC-6`, or *Etc/GMT-8*, which stands for `UTC+8`
  Yes, the UTC timezone offsets are intentionally inverted as GMT entries; you cannot search for something like `UTC+6` or `UTC+5:45`.

A few additional canonical and alias names are supported (e.g., `UTC` or `GMT`) and also some deprecated entries are available (e.g., `PST8PDT`, `US/Hawaii`, `Canada/Central`, or just `Japan`). However, you cannot search for a single timezone abbreviation entry such as `CST` or `PST`. We will discuss later how to deal with timezone abbreviations.

See also `http://en.wikipedia.org/wiki/List_of_tz_database_time_zones` for a list of timezone names.

Accessing the Timezone Database

The IANA timezone database has multiple updates each year to ensure that it is up to date when timezones change so that programs can react accordingly.

The way systems handle the timezone database is implementation-specific. The operating systems have to decide how to provide the necessary data and how to keep it up to date. An update of the timezone database is usually done as part of an operating system update. Such updates generally require a reboot of the machine, and so no C++ application runs during an update.

The C++20 standard provides low-level support to this database, which is used by high-level functions dealing with timezones:

- `std::chrono::`**`get_tzdb_list()`** yields a reference to the timezone database, which is a singleton of type `std::chrono::tzdb_list`.

  It is a list of timezone database entries to be able to support multiple versions of them in parallel.
- `std::chrono::`**`get_tzdb()`** yields a reference to the current timezone database entry of type `std::chrono::tzdb`. This type has multiple members such as the following:
  - `version`, a string for the version of the database entry
  - `zones`, a vector of timezone information of type `std::chrono::time_zone`

  Standard functions that deal with timezones (e.g., `std::chrono::current_zone()` or the constructors of type `std::chrono::zoned_time`) internally use these calls.

  For example, `std::chrono::current_zone()` is a shortcut for:

  ```
  std::chrono::get_tzdb().current_zone()
  ```

When platforms do not provide the timezone database, these functions will throw an exception of type `std::runtime_error`.

For systems that support updating the IANA timezone database without a reboot `std::chrono::`**`reload_tzdb()`** is provided. An update does not delete memory from the old database because the application may still have (`time_zone`) pointers into it. Instead, the new database is atomically pushed onto the front of the timezone database list.

You can check the current version of your timezone database with the string member `version` of the type `tzdb` returned by `get_tzdb()`. For example:

```
std::cout << "tzdb version: " << chr::get_tzdb().version << '\n';
```

Usually, the output is a year and an ascending alphabetic character for the update of that year (e.g., `"2021b"`). `remote_version()` provides the version of the latest available timezone database, which you can use to decide whether to call `reload_tzdb()`.

If a long-running program is using the chrono timezone database but never calls `reload_tzdb()`, that program will not be aware of any updates of the database. It will continue to use whatever version of the database existed when the program first accessed it.

See `http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes#tzdb_manage` for more details and examples about reloading the IANA timezone database for long-running programs.

11.6.3 Using Timezones

To deal with timezones, two types play a basic role:

- `std::chrono::`**`time_zone`**,
  a type that represents a specific timezone.
- `std::chrono::`**`zoned_time`**,
  a type that represents a specific point of time associated with a specific timezone.

Let us look at them in detail.

Type `time_zone`

All possible timezone values are predefined by the IANA timezone database. Therefore, you cannot create a `time_zone` object by just declaring it. The values come from the timezone database and what you are usually dealing with is are pointers to these objects:

- `std::chrono::`**`current_zone()`** yields a pointer to the current timezone.
- `std::chrono::`**`locate_zone(`***name***`)`** yields a pointer to the timezone *name*.
- The timezone database returned by `std::chrono::`**`get_tzdb()`** has non-pointer collections with all timezone entries:
  - Member `zones` has all canonical entries.
  - Member `links` has all alias entries with links to them.

  You can use this to find timezones by characteristics such as their abbreviated name.

For example:

```
auto tzHere = std::chrono::current_zone();        // type const time_zone*
auto tzUTC = std::chrono::locate_zone("UTC");     // type const time_zone*
...
std::cout << tzHere->name() << '\n';
std::cout << tzUTC->name() << '\n';
```

The output depends on your current timezone. For me in Germany, it looks like this:

```
Europe/Berlin
Etc/UTC
```

As you can see, you can search for any entry in the timezone database (such as `"UTC"`) but what you get is the canonical entry of it. For example, `"UTC"` is just a timezone link to `"Etc/UTC"`. If `locate_zone()` finds no corresponding entry with that name, a `std::runtime_error` exception is thrown.

You cannot do much with a `time_zone`. The most important thing is to combine it with a system timepoint or with a local timepoint.

If you output a `time_zone` you get some implementation-specific output only for debugging purposes:

```
std::cout << *tzHere << '\n';     // some implementation-specific debugging output
```

The chrono library also allows you to define and use types for custom timezones.

Type zoned_time

Objects of type `std::chrono::zoned_time` apply timepoints to timezones. You have two options for performing this conversion:

- Apply a system timepoint (a timepoint that belongs to the system clock) to a timezone. In that case, we convert the timepoint of an event to happen simultaneously with the local time of the other timezone.
- Apply a *local timepoint* (a timepoint that belongs to the pseudo clock `local_t`) to a timezone. In that case, we apply a timepoint as it is as local time to another timezone.

In addition, you can convert the point of time of a `zoned_time` to a different timezone by initializing a new `zoned_time` object with another `zoned_time` object.

For example, let us schedule both a local party at every office at 18:00 and a company party over multiple timezones at the end of September 2021:

```
auto day = 2021y/9/chr::Friday[chr::last];                     // last Friday of month
chr::local_seconds tpOfficeParty{chr::local_days{day} - 6h};  // 18:00 the day before
chr::sys_seconds tpCompanyParty{chr::sys_days{day} + 17h};     // 17:00 that day

std::cout << "Berlin Office and Company Party:\n";
std::cout << "  " << chr::zoned_time{"Europe/Berlin", tpOfficeParty} << '\n';
std::cout << "  " << chr::zoned_time{"Europe/Berlin", tpCompanyParty} << '\n';

std::cout << "New York Office and Company Party:\n";
std::cout << "  " << chr::zoned_time{"America/New_York", tpOfficeParty} << '\n';
std::cout << "  " << chr::zoned_time{"America/New_York", tpCompanyParty} << '\n';
```

The output of this code is as follows:

```
Berlin Office and Company Party:
  2021-09-23 18:00:00 CEST
  2021-09-24 19:00:00 CEST
New York Office and Company Party:
  2021-09-23 18:00:00 EDT
  2021-09-24 13:00:00 EDT
```

Details matter when combining timepoints and timezones. For example, consider the following code:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto locTime = chr::zoned_time{chr::current_zone(), sysTp};       // local time
...
std::cout << "sysTp:       " << sysTp << '\n';
std::cout << "locTime:     " << locTime << '\n';
```

First, we initialize `sysTp` as the current system timepoint in seconds and combine this timepoint with the current timezone. The output shows the system and the local timepoint of the same point in time:

```
sysTp:       2021-04-13 13:40:02
locTime:     2021-04-13 15:40:02 CEST
```

Now let us initialize a local timepoint. One way to do this is just to convert a system timepoint to a local timepoint. To do this we need a timezone. If we use the current timezone, the local time is converted to UTC:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto curTp = chr::current_zone()->to_local(sysTp);               // local timepoint
std::cout << "sysTp:       " << sysTp << '\n';
std::cout << "locTp:       " << locTp << '\n';
```

The corresponding output is as follows:

```
sysTp:       2021-04-13 13:40:02
curTp:       2021-04-13 13:40:02
```

However, if we use UTC as the timezone, the local time is *used* as local time without an associated timezone:

```
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto locTp = std::chrono::locate_zone("UTC")->to_local(sysTp);    // use local time as is
std::cout << "sysTp:          " << sysTp << '\n';
std::cout << "locTp:          " << locTp << '\n';
```

According to the output, both timepoints look the same:

```
sysTp:       2021-04-13 13:40:02
locTp:       2021-04-13 13:40:02
```

However, they are not the same. `sysTp` has an associated UTC epoch but `locTp` does not. If we now apply the local timepoint to a timezone, we do not convert; we specify the missing timezone, leaving the time as it is:

```
auto timeFromSys = chr::zoned_time{chr::current_zone(), sysTp};  // converted time
auto timeFromLoc = chr::zoned_time{chr::current_zone(), locTp};  // applied time
std::cout << "timeFromSys: " << timeFromSys << '\n';
std::cout << "timeFromLoc: " << timeFromLoc << '\n';
```

Thus, the output is as follows:

```
timeFromSys: 2021-04-13 15:40:02 CEST
timeFromLoc: 2021-04-13 13:40:02 CEST
```

Now let us combine all four objects with the timezone of New York:

```
std::cout << "NY sysTp:          "
          << std::chrono::zoned_time{"America/New_York", sysTp} << '\n';
std::cout << "NY locTP:          "
          << std::chrono::zoned_time{"America/New_York", locTp} << '\n';
std::cout << "NY timeFromSys: "
          << std::chrono::zoned_time{"America/New_York", timeFromSys} << '\n';
std::cout << "NY timeFromLoc: "
          << std::chrono::zoned_time{"America/New_York", timeFromLoc} << '\n';
```

The output is as follows:

```
NY sysTp:       2021-04-13 09:40:02 EDT
NY locTP:       2021-04-13 13:40:02 EDT
NY timeFromSys: 2021-04-13 09:40:02 EDT
NY timeFromLoc: 2021-04-13 07:40:02 EDT
```

The system timepoint and the local time derived from it both convert the time now to the timezone of New York. As usual, the local timepoint is applied to New York so that we now have the same value as the original time with the timezone removed. `timeFromLoc` is the initial local time 13:40:02 in Central Europe applied to the timezone of New York.

11.6.4 Dealing with Timezone Abbreviations

Because timezone abbreviations might refer to different timezones, you cannot define a unique timezone by its abbreviation. Instead, you have to map the abbreviation to one of multiple IANA timezone entries.

The following program demonstrates this for the timezone abbreviation CST:

lib/chronocst.cpp

```cpp
#include <iostream>
#include <chrono>
using namespace std::literals;

int main(int argc, char** argv)
{
  auto abbrev = argc > 1 ? argv[1] : "CST";

  auto day = std::chrono::sys_days{2021y/1/1};
  auto& db = std::chrono::get_tzdb();

  // print time and name of all timezones with abbrev:
  std::cout << std::chrono::zoned_time{"UTC", day}
            << " maps to these '" << abbrev << "' entries:\n";
  // iterate over all timezone entries:
  for (const auto& z : db.zones) {
    // and map to those using my passed (or default) abbreviation:
    if (z.get_info(day).abbrev == abbrev) {
      std::chrono::zoned_time zt{&z, day};
      std::cout << "  " << zt << "  " << z.name() << '\n';
    }
  }
}
```

The program passing no command-line argument or `"CST"` may have the following output:

```
2021-01-01 00:00:00 UTC maps these 'CST' entries:
  2020-12-31 18:00:00 CST  America/Bahia_Banderas
  2020-12-31 18:00:00 CST  America/Belize
```

```
2020-12-31 18:00:00 CST  America/Chicago
2020-12-31 18:00:00 CST  America/Costa_Rica
2020-12-31 18:00:00 CST  America/El_Salvador
2020-12-31 18:00:00 CST  America/Guatemala
2020-12-31 19:00:00 CST  America/Havana
2020-12-31 18:00:00 CST  America/Indiana/Knox
2020-12-31 18:00:00 CST  America/Indiana/Tell_City
2020-12-31 18:00:00 CST  America/Managua
2020-12-31 18:00:00 CST  America/Matamoros
2020-12-31 18:00:00 CST  America/Menominee
2020-12-31 18:00:00 CST  America/Merida
2020-12-31 18:00:00 CST  America/Mexico_City
...
2020-12-31 18:00:00 CST  America/Winnipeg
2021-01-01 08:00:00 CST  Asia/Macau
2021-01-01 08:00:00 CST  Asia/Shanghai
2021-01-01 08:00:00 CST  Asia/Taipei
2020-12-31 18:00:00 CST  CST6CDT
```

Because `CST` may stand for *Central Standard Time*, *China Standard Time*, or *Cuba Standard Time*, you can see a 14 hour difference between most of the entries for America and China. In addition, Havana in Cuba has a 1 or 13 hour difference to those entries.

Note that the output is significantly smaller when we search for `"CST"` on a day in the summer, because the US entries and Cuba's entry then switch to `"CDT"` (the corresponding daylight saving time). However, we still have some entries because, for example, China and Costa Rica do not have daylight saving time.

Note also that for `CST`, you might find no entry at all, because the timezone database is not available or uses something like `GMT-6` instead of `CST`.

11.6.5 Custom Timezones

The chrono library allows you to use custom timezones. One common example is the need to have a timezone that has an offset from UTC that is not known until run time.

Here is an example that supplies a custom timezone `OffsetZone` that can hold a UTC offset with minutes precision:[7]

lib/offsetzone.hpp

```cpp
#include <chrono>
#include <iostream>
#include <type_traits>

class OffsetZone
{
```

[7] Thanks to Howard Hinnant for providing this example.

```
  private:
    std::chrono::minutes offset;   // UTC offset
  public:
    explicit OffsetZone(std::chrono::minutes offs)
     : offset{offs} {
    }

    template<typename Duration>
    auto to_local(std::chrono::sys_time<Duration> tp) const {
      // define helper type for local time:
      using LT
       = std::chrono::local_time<std::common_type_t<Duration,
                                                    std::chrono::minutes>>;
      // convert to local time:
      return LT{(tp + offset).time_since_epoch()};
    }

    template<typename Duration>
    auto to_sys(std::chrono::local_time<Duration> tp) const {
      // define helper type for system time:
      using ST
       = std::chrono::sys_time<std::common_type_t<Duration,
                                                  std::chrono::minutes>>;
      // convert to system time:
      return ST{(tp - offset).time_since_epoch()};
    }

    template<typename Duration>
    auto get_info(const std::chrono::sys_time<Duration>& tp) const {
      return std::chrono::sys_info{};
    }
};
```

As you can see, all you have to define are conversions between the local time and the system time.

You can use the timezone just like any other `time_zone` pointer:

lib/offsetzone.cpp

```
#include "offsetzone.hpp"
#include <iostream>

int main()
{
  using namespace std::literals;   // for h and min suffix

  // timezone with 3:45 offset:
  OffsetZone p3_45{3h + 45min};

  // convert now to timezone with offset:
  auto now = std::chrono::system_clock::now();
  std::chrono::zoned_time<decltype(now)::duration, OffsetZone*> zt{&p3_45, now};

  std::cout << "UTC:    " << zt.get_sys_time() << '\n';
  std::cout << "+3:45:  " << zt.get_local_time() << '\n';
  std::cout << zt << '\n';
}
```

The program might have the following output:

```
UTC:    2021-05-31 13:01:19.0938339
+3:45:  2021-05-31 16:46:19.0938339
```

11.7 Clocks in Detail

C++20 now supports a couple of clocks. This section discusses the differences between them and how to use special clocks.

11.7.1 Clocks with a Specified Epoch

C++20 now provides the following clocks associated with an epoch (so that they define a unique point in time):

- The **system clock** is the clock of your operating system. Since C++20 it is specified to be Unix Time,[8] which counts time since the epoch January 1, 1970 00:00:00 UTC.

  Leap seconds are handled such that some seconds might take a little bit longer. Therefore, you never have hours with 61 seconds and all years with 365 days have the same number of 31,536,000 seconds.
- The **UTC clock** is the clock that represents the *Coordinated Universal Time*, popularly known as *GMT* (*Greenwich Mean Time*), or *Zulu* time. Local time differs from UTC by the UTC offset of your timezone.

  It uses the same epoch as the system clock (January 1, 1970 00:00:00 UTC).

  Leap seconds are handled such that some minutes might have 61 seconds. For example, there is a timepoint `1972-06-30 23:59:60` because a leap second was added to the last minute of June 1972. Therefore, years with 365 days might sometimes have 31,536,001 or even 31,536,002 seconds.
- The **GPS clock** uses the time of the *Global Positioning System*, which is the atomic time scale implemented by the atomic clocks in the GPS ground control stations and satellites. GPS time starts with the epoch of January 6, 1980 00:00:00 UTC.

  Each minute has 60 seconds, but GPS takes leap seconds into account by switching to the next hour earlier. As a result, GPS is more and more seconds ahead of UTC (or more and more seconds behind before 1980). For example, the timepoint `2021-01-01 00:00:00 UTC` is represented as a GPS timepoint as `2021-01-01 00:00:18 GPS`. In 2021, while writing this book, GPS timepoints are 18 seconds ahead.

  All *GPS years* with 365 days (the difference between midnight of a GPS date and midnight of the GPS date one year later) have the same number of 31,536,000 seconds but might be one or two seconds shorter than the "real" year.
- The **TAI clock** uses *International Atomic Time*, which is the international atomic time scale based on a continuous counting of the SI second. TAI time starts with the epoch of January 1, 1958 00:00:00 UTC.

  As is the case for GPS time, each minute has 60 seconds, and leap seconds are taken into account by switching to the next hour earlier. As a result, TAI is more and more seconds ahead of UTC, but always has a constant offset of 19 seconds to GPS. For example, the timepoint `2021-01-01 00:00:00 UTC` is represented as a TAI timepoint as `2021-01-01 00:00:37 TAI`. In 2021, while writing this book, TAI timepoints are 37 seconds ahead.

[8] For details about Unix time see, for example, `http://en.wikipedia.org/wiki/Unix_time`.

11.7.2 The Pseudo Clock `local_t`

As already introduced, there is a special clock of type `std::chrono::`**`local_t`**. This clock allows us to specify *local timepoints*, which have no timezone (not even UTC) yet. Its epoch is interpreted as "local time," which means you have to combine it with a timezone to know which point in time it represents.

`local_t` is a "pseudo clock" because it does not fulfill all requirements of clocks. In fact, it does not provide a member function `now()`:

```
auto now1 = std::chrono::local_t::now();    // ERROR: now() not provided
```

Instead, you need a system clock timepoint and a timezone. You can then convert the timepoint to a local timepoint using a timezone such as the current timezone or UTC:

```
auto sysNow = chr::system_clock::now();                   // NOW as UTC timepoint
...
chr::local_time now2
  = chr::current_zone()->to_local(sysNow);                // NOW as local timepoint
chr::local_time now3
  = chr::locate_zone("Asia/Tokyo")->to_local(sysNow);     // NOW as Tokyo timepoint
```

Another way to get the same result is to call `get_local_time()` for a zoned time (a timepoint with associated timezone):

```
chr::local_time now4 = chr::zoned_time{chr::current_zone(),
                                       sysNow}.get_local_time();
```

A different approach is to parse a string into a local timepoint:

```
chr::local_seconds tp;   // time_point<local_t, seconds>
std::istringstream{"2021-1-1 18:30"} >> chr::parse(std::string{"%F %R"}, tp);
```

Remember the subtle differences between local timepoints compared to other timepoints:

- A system/UTC/GPS/TAI timepoint represents a specific point in time. Applying it to a timezone will convert the time value it represents.
- A local timepoint represents a local time. Its global point in time becomes clear once it is combined with a timezone.

For example:

```
auto now = chr::current_zone()->to_local(chr::system_clock::now());
std::cout << now << '\n';
std::cout << "Berlin: " << chr::zoned_time("Europe/Berlin", now) << '\n';
std::cout << "Sydney: " << chr::zoned_time("Australia/Sydney", now) << '\n';
std::cout << "Cairo:  " << chr::zoned_time("Africa/Cairo", now) << '\n';
```

This applies the local time of `now` to three different timezones:

```
2021-04-14 08:59:31.640004000
Berlin: 2021-04-14 08:59:31.640004000 CEST
Sydney: 2021-04-14 08:59:31.640004000 AEST
Cairo:  2021-04-14 08:59:31.640004000 EET
```

Note that you cannot use conversion specifiers for the timezone when using a local timepoint:

```
chr::local_seconds tp;    //time_point<local_t, seconds>
...
std::cout << std::format("{:%F %T %Z}\n", tp); // ERROR: invalid format
std::cout << std::format("{:%F %T}\n", tp);    // OK
```

11.7.3 Dealing with Leap Seconds

The previous discussion of clocks with a specified epoch already introduced basic aspects of dealing with leap seconds.

To make the handling of leap seconds more clear, let us iterate over timepoints of a leap second using different clocks:[9]

lib/chronoclocks.cpp

```
#include <iostream>
#include <chrono>

int main()
{
  using namespace std::literals;
  namespace chr = std::chrono;

  auto tpUtc = chr::clock_cast<chr::utc_clock>(chr::sys_days{2017y/1/1} - 1000ms);
  for (auto end = tpUtc + 2500ms; tpUtc <= end; tpUtc += 200ms) {
    auto tpSys = chr::clock_cast<chr::system_clock>(tpUtc);
    auto tpGps = chr::clock_cast<chr::gps_clock>(tpUtc);
    auto tpTai = chr::clock_cast<chr::tai_clock>(tpUtc);
    std::cout << std::format("{:%F %T} SYS  ", tpSys);
    std::cout << std::format("{:%F %T %Z}  ",  tpUtc);
    std::cout << std::format("{:%F %T %Z}  ",  tpGps);
    std::cout << std::format("{:%F %T %Z}\n",  tpTai);
  }
}
```

The program has the following output:

```
2016-12-31 23:59:59.000 SYS  2016-12-31 23:59:59.000 UTC  2017-01-01 00:00:16.000 GPS  2017-01-01 00:00:35.000 TAI
2016-12-31 23:59:59.200 SYS  2016-12-31 23:59:59.200 UTC  2017-01-01 00:00:16.200 GPS  2017-01-01 00:00:35.200 TAI
2016-12-31 23:59:59.400 SYS  2016-12-31 23:59:59.400 UTC  2017-01-01 00:00:16.400 GPS  2017-01-01 00:00:35.400 TAI
2016-12-31 23:59:59.600 SYS  2016-12-31 23:59:59.600 UTC  2017-01-01 00:00:16.600 GPS  2017-01-01 00:00:35.600 TAI
2016-12-31 23:59:59.800 SYS  2016-12-31 23:59:59.800 UTC  2017-01-01 00:00:16.800 GPS  2017-01-01 00:00:35.800 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.000 UTC  2017-01-01 00:00:17.000 GPS  2017-01-01 00:00:36.000 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.200 UTC  2017-01-01 00:00:17.200 GPS  2017-01-01 00:00:36.200 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.400 UTC  2017-01-01 00:00:17.400 GPS  2017-01-01 00:00:36.400 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.600 UTC  2017-01-01 00:00:17.600 GPS  2017-01-01 00:00:36.600 TAI
2016-12-31 23:59:59.999 SYS  2016-12-31 23:59:60.800 UTC  2017-01-01 00:00:17.800 GPS  2017-01-01 00:00:36.800 TAI
2017-01-01 00:00:00.000 SYS  2017-01-01 00:00:00.000 UTC  2017-01-01 00:00:18.000 GPS  2017-01-01 00:00:37.000 TAI
2017-01-01 00:00:00.200 SYS  2017-01-01 00:00:00.200 UTC  2017-01-01 00:00:18.200 GPS  2017-01-01 00:00:37.200 TAI
2017-01-01 00:00:00.400 SYS  2017-01-01 00:00:00.400 UTC  2017-01-01 00:00:18.400 GPS  2017-01-01 00:00:37.400 TAI
```

[9] Thanks to Howard Hinnant for providing this example.

The leap second we look at here is the last leap second we had when this book was written (we do not know in advance when leap seconds will happen in the future). We print the timepoint with the corresponding timezone (for system timepoints, we print SYS instead of its default timezone UTC). You can observe the following:

- During the leap second:
  - The UTC time uses the value 60 as seconds.
  - The system clock uses the last representable value of sys_time prior to the insertion of the leap second. That behavior is guaranteed by the C++ standard.
- Before the leap second:
  - The GPS time is 17 seconds ahead of the UTC time.
  - The TAI time is 36 seconds ahead of the UTC time (as always, 19 seconds ahead of the GPS time).
- After the leap second:
  - The GPS time is 18 seconds ahead of the UTC time.
  - The TAI time is 37 seconds ahead of the UTC time (still 19 seconds ahead of the GPS time).

11.7.4 Conversions between Clocks

You can convert timepoint between clocks provided such a conversion makes sense. For this purpose, the chrono library provides a clock_cast<>. It is defined in a way that you can only convert between the timepoints of clocks that have a specified stable epoch (sys_time<>, utc_time<>, gps_time<>, tai_time<>) and filesystem timepoints.

The cast needs the destination clock and you can optionally pass a different duration.

The following program creates the output of a UTC leap second to a couple of other clocks:

lib/chronoconv.cpp

```cpp
#include <iostream>
#include <sstream>
#include <chrono>

int main()
{
  namespace chr = std::chrono;

  // initialize a utc_time<> with a leap second:
  chr::utc_time<chr::utc_clock::duration> tp;
  std::istringstream{"2015-6-30 23:59:60"}
    >> chr::parse(std::string{"%F %T"}, tp);

  // convert it to other clocks and print that out:
  auto tpUtc = chr::clock_cast<chr::utc_clock>(tp);
  std::cout << "utc_time:  " << std::format("{:%F %T %Z}", tpUtc) << '\n';
  auto tpSys = chr::clock_cast<chr::system_clock>(tp);
  std::cout << "sys_time:  " << std::format("{:%F %T %Z}", tpSys) << '\n';
  auto tpGps = chr::clock_cast<chr::gps_clock>(tp);
  std::cout << "gps_time:  " << std::format("{:%F %T %Z}", tpGps) << '\n';
  auto tpTai = chr::clock_cast<chr::tai_clock>(tp);
  std::cout << "tai_time:  " << std::format("{:%F %T %Z}", tpTai) << '\n';
  auto tpFile = chr::clock_cast<chr::file_clock>(tp);
  std::cout << "file_time: " << std::format("{:%F %T %Z}", tpFile) << '\n';
}
```

The program has the following output:

```
utc_time:  2015-06-30 23:59:60.0000000 UTC
sys_time:  2015-06-30 23:59:59.9999999 UTC
gps_time:  2015-07-01 00:00:16.0000000 GPS
tai_time:  2015-07-01 00:00:35.0000000 TAI
file_time: 2015-06-30 23:59:59.9999999 UTC
```

Note that for all of these clocks, we have pseudo timezones for formatted output.

The rules of conversions are as follows:

- Any conversion from a local timepoint adds just the epoch. The time value remains the same.
- Conversions between UTC, GPS, and TAI timepoints add or subtract the necessary offsets.
- Conversions between UTC and system time do not change the time value except that for the timepoint of a UTC leap second, the last timepoint ahead is used as the system time.

Conversions from local timepoints to other clocks are also supported. However, for conversions to local timepoints, you have to use `to_local()` (after converting to a system timepoint).

Conversions to or from timepoints of the `steady_clock` are not supported.

Clock Conversions Internally

Under the hood, `clock_cast<>` is a "two-hub spoke and wheel system."[10] The two hubs are `system_clock` and `utc_clock`. Every convertible clock has to convert to and from one of these hubs (but not both). The conversions *to* the hubs are done with the `to_sys()` or `to_utc()` static member functions of the clocks. For conversions *from* the hubs, `from_sys()` or `from_utc()` are provided. `clock_cast<>` strings these member functions together to convert from any clock to any other clock.

Clocks that cannot deal with leap seconds should convert to/from `system_clock`. For example, `utc_clock` and `local_t` provide `to_sys()`.

Clocks that can deal with leap seconds in some way (which does not necessarily mean that they have a leap second value) should convert to/from `utc_clock`. This applies to `gps_clock` and `tai_clock`, because even though GPS and TAI do not have leap seconds, they have unique, bidirectional mappings to UTC leap seconds.

For the `file_clock`, it is implementation-specific whether conversions to/from `system_clock` or `utc_clock` are provided.

[10] Thanks to Howard Hinnant for pointing this out.

11.7.5 Dealing with the File Clock

The clock `std::chrono::`**`file_clock`** is the clock the filesystem library uses for timepoints of filesystem entries (files, directories, etc.). It is an implementation-specific clock type that reflects the resolution and range of time values of the filesystem.

For example, you can use the file clock to update the time of last access to a file as follows:

```
// touch file with path p (update last write access to file):
std::filesystem::last_write_time(p,
                                 std::chrono::file_clock::now());
```

You were also able to use the clock of filesystem entries in C++17 by using the type `std::filesystem::file_time_type`:

```
std::filesystem::last_write_time(p,
                                 std::filesystem::file_time_type::clock::now());
```

Since C++20, the filesystem type name `file_time_type` is defined as follows:

```
namespace std::filesystem {
  using file_time_type = chrono::time_point<chrono::file_clock>;
}
```

In C++17, you could only use an unspecified *trivial-clock*.

For timepoints of the filesystem, the type `file_time` is also defined now:

```
namespace std::chrono {
  template<typename Duration>
  using file_time = time_point<file_clock, Duration>;
}
```

A type like `file_seconds` (as the other clocks have) is not defined.

The new definition of the `file_time` type now allows programmers to portably convert filesystem timepoints to system timepoints. For example, you can print the time when a passed file was accessed last as follows:

```
void printFileAccess(const std::filesystem::path& p)
{
  std::cout << "\"" << p.string() << "\":\n";

  auto tpFile = std::filesystem::last_write_time(p);
  std::cout << std::format("  Last write access: {0:%F} {0:%X}\n", tpFile);

  auto diff = std::chrono::file_clock::now() - tpFile;
  auto diffSecs = std::chrono::round<std::chrono::seconds>(diff);
  std::cout << std::format("  It is {} old\n", diffSecs);
}
```

This code might output:

```
"chronoclocks.cpp":
  Last write access: 2021-07-12 16:50:08
  It is 18s old
```

If you were to use the default output operator of timepoints, which prints subseconds according to the granularity of the file clock:

```
std::cout << "  Last write access: " << diffSecs << '\n';
```

the output might be as follows:

```
  Last write access: 2021-07-12 16:50:08.3680536
```

To deal with the file access time as system or local time, you have to use a `clock_cast<>()` (which, internally, might call the static `file_clock` member functions `to_sys()` or `to_utc()`). For example:

```
auto tpFile = std::filesystem::last_write_time(p);
auto tpSys = std::chrono::file_clock::to_sys(tpFile);
auto tpSys = std::chrono::clock_cast<std::chrono::system_clock>(tpFile);
```

11.8 Other New Chrono Features

In addition to what I have described so far, the new chrono library adds the following features:

- To check whether a type is a clock, a new type trait `std::chrono::is_clock<>` is provided together with its corresponding variable template `std::chrono::is_clock_v<>`. For example:

  ```
  std::chrono::is_clock_v<std::chrono::system_clock>    // true
  std::chrono::is_clock_v<std::chrono::local_t>         // false
  ```

  The pseudo clock `local_t` yields `false` here because it does not provide the member function `now()`.
- For durations and timepoints, `operator<=>` is defined.

11.9 Afternotes

The chrono library was developed by Howard Hinnant. When the basic part for durations and timepoints was standardized with C++11, the plan was already to extend this with support for dates, calendars, and timezones.

The chrono extension for C++20 was first proposed by Howard Hinnant in http://wg21.link/p0355r0. The finally accepted wording for this extension was formulated by Howard Hinnant and Tomasz Kaminski in http://wg21.link/p0355r7.

A few minor fixes were added by Howard Hinnant in http://wg21.link/p1466r3 and by Tomasz Kaminski in http://wg21.link/p1650r0. The finally accepted wording for the integration of the chrono library with the formatting library was formulated by Victor Zverovich, Daniela Engert, and Howard Hinnant in http://wg21.link/p1361r2.

After finishing C++20, a few fixes were applied to fix the chrono extensions of the C++20 standard:

- http://wg21.link/p2372 clarified the behavior of locale-dependent formatted output.
- http://wg21.link/lwg3554 ensures that you can pass string literals as format to `parse()`.

This page is intentionally left blank

Chapter 12

`std::jthread` and Stop Tokens

C++20 introduces a new type for representing threads: `std::jthread`. It fixes a severe design problem of `std::thread` and benefits from a new feature for signaling cancellation.

This chapter explains both the feature for signaling cancellation in asynchronous scenarios and the new class `std::jthread`.

12.1 Motivation for `std::jthread`

C++11 introduced the type `std::thread` that maps one-to-one with threads as provided by the operating system. However, the type has a severe design flaw: it is not an RAII type.

Let us look at why this is a problem and how the new thread type solves this problem.

12.1.1 The Problem of `std::thread`

`std::thread` requires that at the end of its lifetime if representing a running thread, either `join()` (to wait for the end of the thread) or `detach()` (to let the thread run in the background) is called. If neither has been called, the destructor immediately causes an abnormal program termination (causing a core dump on some systems). For this reason, the following code is usually an error (unless you do not care about abnormal program termination):

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ...   // neither t.join() nor t.detach() called

} // std::terminate() called
```

When the destructor of `t` representing the running thread is called without having called `join()` or `detach()`, the program calls `std::terminate()`, which calls `std::abort()`.

Even when calling `join()` to wait for the running thread to end, you still have a significant problem:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ...     // calls std::terminate() on exception
  // wait for tasks to finish:
  t.join();
  ...
}
```

This code may also cause an abnormal program termination because `t.join()` is not called when an exception occurs in `foo()` between the start of the thread and the call of `join()` (or if the control flow never reaches the call to `join()` for any other reason).

The way you have to program this is as follows:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  try {
    ...               // might throw an exception
  }
  catch (...) {     // if we have an exception
    // clean up the thread started:
    t.join();       // - wait for thread (blocks until done)
    throw;          // - and rethrow the caught exception
  }
  // wait for thread to finish:
  t.join();
  ...
}
```

Here, we react to an exception without resolving it by making sure that `join()` is called when we leave the scope. Unfortunately, this might block (forever). However, calling `detach()` is also a problem because the thread continues in the background of the program, using CPU time and resources that might now be destroyed.

If you use multiple threads in more complex contexts, the problem gets even worse and creates really nasty code. For example, when starting just two threads, you have to program something like this:

```
void foo()
{
  ...
  // start thread calling task1() with name and val as arguments:
  std::thread t1{task1, name, val};
  std::thread t2;
  try {
    // start thread calling task2() with name and val as arguments:
    t2 = std::thread{task2, name, val};
    ...
  }
  catch (...) { // if we have an exception
    // clean up the threads started:
    t1.join();              // wait for first thread to end
    if (t2.joinable()) {    // if the second thread was started
      t2.join();            // - wait for second thread to end
    }
    throw;          // and rethrow the caught exception
  }
  // wait for threads to finish:
  t1.join();
  t2.join();
  ...
}
```

On one hand, after starting the first thread, starting the second thread might throw, so starting the second thread has to happen in the `try` clause. On the other hand, we want to use and `join()` both threads in the same scope. To fulfill both requirements, we have to forward declare the second thread and move assign it in the `try` clause of the first thread. In addition, on an exception, we have to check whether the second thread was started or not because calling `join()` for a thread object that has no associated thread causes another exception.

An additional problem is that calling `join()` for both threads might take a significant amount of time (or even take forever). Note that you cannot "kill" threads that have been started. Threads are not processes. A thread can only end by ending itself or ending the program as a whole.

Therefore, before calling `join()`, you should make sure that the thread you wait for will cancel its execution. However, with `std::thread`, there is no mechanism for that. You have to implement the request for cancellation and the reaction to it yourself.

12.1.2 Using `std::jthread`

`std::jthread` solves these problems. First, it is an RAII type. The destructor calls `join()` if the thread is joinable (the "j" stands for "joining"). Thus, the complex code above simply becomes:

```
  void foo()
  {
    ...
    // start thread calling task1() with name and val as arguments:
    std::jthread t1{task1, name, val};
    // start thread calling task2() with name and val as arguments:
    std::jthread t2{task2, name, val};
    ...
    // wait for threads to finish:
    t1.join();
    t2.join();
    ...
  }
```

By simply using `std::jthread` instead of `std::thread`, the danger of causing an abnormal program termination is no longer present and no exception handling is necessary. To support switching to the class `std::jthread` as easily as possible, the class provides the same API as `std::thread`, including:

- Using the same header file `<thread>`
- Returning a `std::thread::id` when calling `get_id()` (the type `std::jthread::id` is just an alias type)
- Providing the `static` member `hardware_concurrency()`

That means: **simply replace `std::thread` by `std::jthread` and recompile** and your code becomes safer (if you did not already implemented the exception handling yourself).[1]

12.1.3 Stop Tokens and Stop Callbacks

The class `std::jthread` does even more: it provides a mechanism for signaling cancellation using stop tokens, which are used by the destructors of `jthreads` before they call `join()`. However, the callable (function, function object, or lambda) started by the thread must support this request:

- If the callable only provides parameters for all passed arguments, the request to stop would be ignored:

  ```
  void task (std::string s, double value)
  {
    ...   // join() waits until this code ends
  }
  ```

- To react to the stop request, the callable can add a new optional first parameter of type `std::stop_token` and check from time to time whether a stop was requested:

  ```
  void task (std::stop_token st,
             std::string s, double value)
  {
    while (!st.stop_requested()) {   // stop requested (e.g., by the destructor)?
      ...   // ensure we check from time to time
    }
  }
  ```

This means that `std::jthread` provides a *cooperative* mechanism to signal that a thread should no longer run. It is “cooperative” because the mechanism does not kill the running thread (killing threads might easily leave a program in a corrupt state and is therefore not supported by C++ threads at all). To honor a request to stop, the started thread has to declare the stop token as an additional first parameter and use it to check from time to time whether it should continue to run.

[1] You might wonder why we did not just fix `std::thread` instead of introducing a new type `std::jthread`. The reason is backward compatibility. There might be a few applications that want to terminate the program when leaving the scope of a running thread. And for some new functionality discussed next, we would also break binary compatibility.

You can also manually request a stop for a jthread that has started. For example:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::jthread t{task, name, val};
  ...
    if (...) {
      t.request_stop();    // explicitly request task() to stop its execution
    }
  ...
  // wait for thread to finish:
  t.join();
  ...
}
```

In addition, there is another way to react to a stop request: you can register callbacks for a stop token, which are automatically called when a stop is requested. For example:

```
void task (std::stop_token st,
           std::string s, double value)
{
  std::stop_callback cb{st, [] {
                                ...   // called on a stop request
                              }};
  ...
}
```

In this case, a request to stop the thread performing `task()` (whether it is an explicit call of `request_stop()` or caused by the destructor) calls the lambda you have registered as the stop callback. Note that the callback is usually called by the thread requesting the stop.

At the end of the lifetime of the stop callback `cb`, the destructor unregisters the callback automatically so that it will no longer be called if a stop is signaled afterwards. You can register an arbitrary number of callables (functions, function objects, or lambdas) that way.

Note that the stop mechanism is more flexible than it looks at first:

- You can pass around handles to request a stop and tokens to check for a requested stop.
- There is support for condition variables so that a signaled stop can interrupt a wait there.
- You can use the mechanism to request and check for stops independently of `std::jthread`.

There are also no lifetime constraints between the thread, its stop source, stop tokens, and stop callbacks. The place where the stop state is stored is allocated on the heap. The memory for the stop state is released when the thread and the last stop source, stop token, or stop callback using this state is destroyed.

In the following sections, we discuss the mechanism for requesting stops and its application in threads. Afterwards, we discuss the underlying stop token mechanism.

12.1.4 Stop Tokens and Condition Variables

When a stop is requested, a thread might be blocked by waiting for the notification for a condition variable (an important scenario to avoid active polling). The callback interface of stop tokens also supports this case. You can call `wait()` for a condition variable with a passed stop token so that the wait is suspended when a stop is requested. Note that for technical reasons, you have to use the type `std::condition_variable_any` for the condition variable.

Here is an example that demonstrates using stop tokens with condition variables:

lib/stopcv.cpp

```
#include <iostream>
#include <queue>
#include <thread>
#include <stop_token>
#include <mutex>
#include <condition_variable>
using namespace std::literals;   // for duration literals

int main()
{
  std::queue<std::string> messages;
  std::mutex messagesMx;
  std::condition_variable_any messagesCV;

  // start thread that prints messages that occur in the queue:
  std::jthread t1{[&] (std::stop_token st) {
                    while (!st.stop_requested()) {
                      std::string msg;
                      {
                        // wait for the next message:
                        std::unique_lock lock(messagesMx);
                        if (!messagesCV.wait(lock, st,
                                             [&] {
                                               return !messages.empty();
                                             })) {
                          return;   // stop requested
                        }
                        // retrieve the next message out of the queue:
                        msg = messages.front();
                        messages.pop();
                      }

                      // print the next message:
                      std::cout << "msg: " << msg << std::endl;
                    }
                  }};
  // store 3 messages and notify one waiting thread each time:
  for (std::string s : {"Tic", "Tac", "Toe"}) {
    std::scoped_lock lg{messagesMx};
    messages.push(s);
    messagesCV.notify_one();
  }

  // after some time
  // - store 1 message and notify all waiting threads:
  std::this_thread::sleep_for(1s);
  {
    std::scoped_lock lg{messagesMx};
    messages.push("done");
    messagesCV.notify_all();
  }

  // after some time
  // - end program (requests stop, which interrupts wait())
  std::this_thread::sleep_for(1s);
}
```

We start one thread that loops over waiting for a queue of messages not to be empty and prints messages if there are any:

```
while (!st.stop_requested()) {
  std::string msg;
  {
    // wait for the next message:
    std::unique_lock lock(messagesMx);
    if (!messagesCV.wait(lock, st,
                         [&] {
                           return !messages.empty();
                         })) {
      return;   // stop requested
    }
    // retrieve the next message out of the queue:
    msg = messages.front();
    messages.pop();
  }

  // print the next message:
  std::cout << "msg: " << msg << std::endl;
}
```

The condition variable messagesCV is of type std::condition_variable_any:

```
std::condition_variable_any messagesCV;
```

That allows us to call wait() with a stop token, where we usually pass the stop token signaling to stop the thread. As a result, the waiting might end now for one of two reasons:

- There was a notification (that the queue is no longer empty)
- A stop was requested

The return value of wait() yields whether the condition is met. If it yields false, the reason to end wait() was a requested stop, meaning that we can react accordingly (here, we stop the loop).

Table condition_variable_any *member functions for stop tokens* lists the new member functions of type std::condition_variable_any for stop tokens using a lock guard *lg*.

Operation	**Effect**
cv.**wait**(*lg*, *st*, *pred*)	Waits for notification with *pred* being true or a stop requested for *st*
cv.**wait_for**(*lg*, *dur*, *st*, *pred*)	Waits at most duration *dur* for notification with *pred* being true or a stop requested for *st*
cv.**wait_until**(*lg*, *tp*, *st*, *pred*)	Waits until timepoint *tp* for notification with *pred* being true or a stop requested for *st*

Table 12.1. condition_variable_any *member functions for stop tokens*

Stop token support for other blocking functions is not supported yet.

12.2 Stop Sources and Stop Tokens

C++20 provides stop tokens not only for threads. It is a general-purpose mechanism to asynchronously request to stop with various ways to react to this request.

The basic mechanism is as follows:

- The C++20 standard library allows us to establish a *shared stop state*. By default, a stop is not signaled.
- *Stop sources* of type std::stop_source can *request* a stop in their associated shared stop state.
- *Stop tokens* of type std::stop_token can be used to react to a stop request in their associated shared stop state. You can actively poll whether there is a stop requested or register a callback of type std::stop_callback, which will be called when a stop is/was requested.
- Once a stop request has been made, it cannot be withdrawn (a subsequent stop request has no effect).
- Stop sources and stop tokens can be copied and moved around to allow code to signal or react to a stop at multiple locations. Copying a source or token is relatively cheap so you usually pass them by value to avoid any lifetime issues.

  However, copying is not as cheap as passing an integral value or raw pointer around. It is more like passing a shared pointer. If you frequently pass them to a sub-function, it might be better to pass them by reference.
- The mechanism is thread safe and can be used in concurrent situations. Stop requests, checks for requested stops, and calls to register or unregister callbacks are properly synchronized, and the associated shared stop state is automatically destroyed when the last user (stop source, stop token, or stop callback) is destroyed.

The following example shows how to establish both a stop source and a stop token:

```
#include <stop_token>
...

// create stop_source and stop_token:
std::stop_source ssrc;                      // creates a shared stop state
std::stop_token stok{ssrc.get_token()};     // creates a token for the stop state
```

The first step is simply to create the `stop_source` object, which provides the API to request a stop. The constructor also creates the associated shared stop state. Then, you can ask the stop source for the `stop_token` object, which provides the API to react to a stop request (by polling or registering callbacks).

You can then pass the token (and/or the source) to locations/threads to establish the asynchronous communication between the places that might request a stop and those that might react to a stop.

There is no other way to create a stop token with an associated shared stop state. The default constructor of a stop token has no associated stop state.

12.2.1 Stop Sources and Stop Tokens in Detail

Let us look at the APIs of stop sources, stop tokens, and stop callbacks in detail. All types are declared in the header file `<stop_token>`.

Stop Sources in Detail

Table *Operations of objects of class* `stop_source` lists the API of `std::stop_source`.

The constructor usually allocates memory on the heap for the *stop state*, which is used by all stop tokens and stop callbacks. There is no way to specify a different location with an allocator.

Note that there are no lifetime constraints between stop sources, stop tokens, and stop callbacks. The memory for the stop state is automatically released when the last stop source, stop token, or stop callback using this state is destroyed.

To enable you to create stop sources with no associated stop state (which might be useful because the stop state needs resources), you can create a stop source with a special constructor and assign a stop source later:

```
std::stop_source ssrc{std::nostopstate};    // no associated shared stop state
...
ssrc = std::stop_source{};                  // assign new shared stop state
```

Stop Tokens in Detail

Table *Operations of objects of class* `stop_token` lists the API of `std::stop_token`.

Note that `stop_possible()` yields `false` signals whether stops can still occur. It yields `false` in two situations:

- If there is no associated stop state
- If there is a stop state, but there is no longer a stop source and a stop was never requested

This can be used to avoid defining reactions to stops that can never occur.

Operation	Effect
stop_source *s*	Default constructor; creates a stop source with associated stop state
stop_source *s*{nostopstate}	Creates a stop source with no associated stop state
stop_source *s*{*s2*}	Copy constructor; creates a stop source that shares the associated stop state of *s2*
stop_source *s*{move(*s2*)}	Move constructor; creates a stop source that gets the associated stop state of *s2* (*s2* no longer has an associated stop state)
s.~***stop_source***()	Destructor; destroys the associated shared stop state if this is the last user of it
s = *s2*	Copy assignment; copy assigns the state of *s2* so that *s* now also shares the stop state of *s2* (any former stop state of *s* is released)
s = move(*s2*)	Move assignment; move assigns the state of *s2* so that *s* now shares the stop state of *s2* (*s2* no longer has a stop state and any former stop state of *s* is released)
s.**get_token()**	Yields a stop_token for the associated stop state (returns a stop token with no associated stop state if there is no stop state to share)
s.**request_stop()**	Requests a stop on the associated stop state if any of it is not done yet (returns whether a stop was requested)
s.stop_possible()	Yields whether *s* has an associated stop state
s.stop_requested()	Yields whether *s* has an associated stop state for which a stop was requested
s1 == *s2*	Yields whether *s1* and *s2* share the same stop state (or both share none)
s1 != *s2*	Yields whether *s1* and *s2* do not share the same stop state
s1.swap(*s2*)	Swaps the states of *s1* and *s2*
swap(*s1*, *s2*)	Swaps the states of *s1* and *s2*

Table 12.2. Operations of objects of class stop_source

12.2.2 Using Stop Callbacks

A stop callback is an object of the RAII type std::stop_callback. The constructor registers a callable (function, function object, or lambda) to be called when a stop is requested for a specified stop token:

```
void task(std::stop_token st)
{
  // register temporary callback:
  std::stop_callback cb{st, []{
    std::cout << "stop requested\n";
    ...
  }};
  ...
} // unregisters callback
```

Operation	**Effect**
stop_token `t`	Default constructor; creates a stop token with no associated stop state
stop_token `t{t2}`	Copy constructor; creates a stop token that shares the associated stop state of `t2`
stop_token `t{move(t2)}`	Move constructor; creates a stop token that gets the associated stop state of `t2` (`t2` no longer has an associated stop state)
`t.~`***stop_token***`()`	Destructor; destroys the associated shared stop state if this is the last user of it
`t = t2`	Copy assignment; copy assigns the state of `t2` so that `t` now also shares the stop state of `t2` (any former stop state of `t` is released)
`t = move(t2)`	Move assignment; move assigns the state of `t2` so that `t` now shares the stop state of `t2` (`t2` no longer has a stop state and any former stop state of `t` is released)
`t.stop_possible()`	Yields whether `t` has an associated stop state and a stop was or can (still) be requested
`t.`**`stop_requested()`**	Yields whether `t` has an associated stop state for which a stop was requested
`t1 == t2`	Yields whether `t1` and `t2` share the same stop state (or both share none)
`t1 != t2`	Yields whether `t1` and `t2` do not share the same stop state
`t1.swap(t2)`	Swaps the states of `t1` and `t2`
`swap(t1, t2)`	Swaps the states of `t1` and `t2`
stop_token `cb{t, f}`	Registers `cb` as stop callback of `t` calling `f`

Table 12.3. Operations of objects of class `stop_token`

Assume we have created the shared stop state and create an asynchronous situation where one thread might request a stop and another thread might run `task()`. We might create this situation with code like the following:

```
// create stop_source with associated stop state:
std::stop_source ssrc;

// register/start task() and pass the corresponding stop token to it:
registerOrStartInBackgound(task, ssrc.get_token());
...
```

The function *registerOrStartInBackgound*`()` could start `task()` immediately, or start `task()` later by calling `std::async()` or initializing a `std::thread`, call a coroutine, or register an event handler.

Now, whenever we request a stop:

```
ssrc.request_stop();
```

one of the following things can happen:

- If `task()` has been started with its callback was initialized, is still running, and the destructor of the callback has not been called yet, the registered callable is called immediately in the thread where `request_stop()` was called. `request_stop()` blocks until all registered callables have been called. The order of the calls is not defined.
- If `task()` has not been started (or its callback has not been initialized yet), `request_stop()` changes the stop state to signal that a stop was requested and returns. If `task()` is started later and the callback is initialized, the callable is called immediately in the thread where the callback is initialized. The constructor of the callback blocks until the callable returns.
- If `task()` has already finished (or at least the destructor of the callback has been called), the callable will never be called. The end of the lifetime of the callback signals that there is no longer a need to call the callable.

These scenarios are carefully synchronized. If we are in the middle of initializing a `stop_callback` so that the callable is registered, one of the above scenarios will happen. The same applies if a stop is requested while a callable is being unregistered due to destroying a stop callback. If the callable has already been started by another thread, the destructor blocks until the callable has finished.

For your programming logic, this means that from the moment you initialize the callback until the end of its destruction, the registered callable might be called. Up to the end of the constructor, the callback runs in the thread of the initialization; afterwards it runs in the thread requesting the stop. The code that requests a stop *might* immediately call the registered callable, it might call it later (if the callback is initialized later), or might never call it (if it is too late to call the callback).

For example, consider the following program:

lib/stop.cpp

```
#include <iostream>
#include <stop_token>
#include <future>      // for std::async()
#include <thread>      // for sleep_for()
#include <syncstream> // for std::osyncstream
#include <chrono>
using namespace std::literals;    // for duration literals

auto syncOut(std::ostream& strm = std::cout) {
  return std::osyncstream{strm};
}

void task(std::stop_token st, int num)
{
  auto id = std::this_thread::get_id();
  syncOut() << "call task(" << num << ")\n";
  // register a first callback:
  std::stop_callback cb1{st, [num, id]{
    syncOut() << "- STOP1 requested in task(" << num
              << (id == std::this_thread::get_id() ? ")\n"
                                                    : ") in main thread\n");
  }};
  std::this_thread::sleep_for(9ms);

  // register a second callback:
  std::stop_callback cb2{st, [num, id]{
    syncOut() << "- STOP2 requested in task(" << num
              << (id == std::this_thread::get_id() ? ")\n"
                                                    : ") in main thread\n");
  }};
  std::this_thread::sleep_for(2ms);
}

int main()
{
  // create stop_source and stop_token:
  std::stop_source ssrc;
  std::stop_token stok{ssrc.get_token()};

  // register callback:
  std::stop_callback cb{stok, []{
    syncOut() << "- STOP requested in main()\n" << std::flush;
  }};

  // in the background call task() a bunch of times:
  auto fut = std::async([stok] {
                          for (int num = 1; num < 10; ++num) {
                            task(stok, num);
                          }
                        });

  // after a while, request stop:
  std::this_thread::sleep_for(120ms);
  ssrc.request_stop();
}
```

Note that we use synchronized output streams to ensure that the print statements of the different threads are synchronized line by line.

For example, the output might be as follows:

```
call task(1)
call task(2)
...
call task(7)
call task(8)
- STOP2 requested in task(8) in main thread
- STOP1 requested in task(8) in main thread
- STOP requested in main()
call task(9)
- STOP1 requested in task(9)
- STOP2 requested in task(9)
```

Or it might also be like this:

```
call task(1)
call task(2)
call task(3)
call task(4)
- STOP2 requested in task(4) in main thread
call task(5)
- STOP requested in main()
- STOP1 requested in task(5)
- STOP2 requested in task(5)
call task(6)
- STOP1 requested in task(6)
- STOP2 requested in task(6)
call task(7)
- STOP1 requested in task(7)
- STOP2 requested in task(7)
...
```

Or it might look like this:

```
call task(1)
call task(2)
call task(3)
call task(4)
- STOP requested in main()
call task(5)
- STOP1 requested in task(5)
- STOP2 requested in task(5)
call task(6)
- STOP1 requested in task(6)
- STOP2 requested in task(6)
...
```

It might even just be:

```
call task(1)
call task(2)
...
call task(8)
call task(9)
- STOP requested in main()
```

Without using `syncOut()`, the output might even have interleaved characters because the output from the main thread and the thread running `task()` might be completely mixed.

Stop Callbacks in Detail

The type of stop callback, `stop_callback`, is a class template with a very limited API. Effectively, it only provides a constructor to register a callable for a stop token and a destructor that unregisters the callable. Copying and moving are deleted and no other member function is provided.

The template parameter is the type of the callable and is usually deduced when the constructor is initialized:

```
auto func = [] { ... };

std::stop_callback cb{myToken, func};   // deduces stop_callback<decltype(func)>
```

The only public member besides constructor and destructor is `callback_type`, which is the type of the stored callable.

The constructors accept both lvalues (objects with a name) and rvalues (temporary objects or objects marked with `std::move()`):

```
auto func = [] { ... };

std::stop_callback cb1{myToken, func};              // copies func
std::stop_callback cb2{myToken, std::move(func)};   // moves func
std::stop_callback cb3{myToken, [] { ... }};        // moves the lambda
```

12.2.3 Constraints and Guarantees of Stop Tokens

The feature to deal with stop requests is pretty robust regarding a couple of scenarios that might occur in asynchronous contexts. However, it cannot avoid every pitfall.

The C++20 standard library guarantees the following:

- All `request_stop()`, `stop_requested()`, and `stop_possible()` calls are synchronized.
- The callback registration is guaranteed to be performed atomically. If there is a concurrent call to `request_stop()` from another thread, then either the current thread will see the request to stop and immediately invoke the callback on the current thread, or the other thread will see the callback registration and will invoke the callback before returning from `request_stop()`.
- The callable of a `stop_callback` is guaranteed not to be called after the destructor of the `stop_callback` returns.
- The destructor of a callback waits for its callable to finish if it is just called by another thread (it does not wait for other callables to finish).

However, note the following constraints:

- A callback should not throw. If an invocation of its callable exits via an exception, then `std::terminate()` is called.
- Do not destroy a callback in its own callable. The destructor does not wait for the callback to finish.

12.3 std::jthread in Detail

Table *Operations of objects of class* `jthread` lists the API of `std::thread`. The column **Diff** notes member functions that **Mod**ified the behavior or are **New** compared to `std::thread`.

Operation	Effect	Diff
jthread *t*	Default constructor; creates a *nonjoinable* thread object	
jthread *t*{*f*,...}	Creates an object representing a new thread that calls *f* (with additional args) or throws `std::system_error`	
jthread *t*{*rv*}	Move constructor; creates a new thread object, which gets the state of *rv*, and makes *rv* *nonjoinable*	
t.~***jthread***()	Destructor; calls `request_stop()` and `join()` if the object is *joinable*	Mod
t = *rv*	Move assignment; move assigns the state of *rv* to *t* (calls `request_stop()` and `join()` if *t* is *joinable*)	Mod
t.`joinable()`	Yields `true` if *t* has an associated thread (is *joinable*)	
t.`join()`	Waits for the associated thread to finish and makes the object *nonjoinable* (throws `std::system_error` if the thread is not *joinable*)	
t.`detach()`	Releases the association of *t* to its thread while the thread continues and makes the object *nonjoinable* (throws `std::system_error` if the thread is not *joinable*)	
t.**`request_stop()`**	Requests a stop on the associated stop token	New
t.**`get_stop_source()`**	Yields an object to request a stop from	New
t.**`get_stop_token()`**	Yields an object to check for a requested stop	New
t.`get_id()`	Returns a unique thread ID of the member type `id` if *joinable* or a default constructed ID if not	
t.`native_handle()`	Returns a platform-specific member type `native_handle_type` for a non-portable handling of the thread	
t1.`swap(`*t2*`)`	Swaps the states of *t1* and *t2*	
`swap(`*t1*, *t2*`)`	Swaps the states of *t1* and *t2*	
`hardware_concurrency()`	Static function with a hint about possible hardware threads	

Table 12.4. Operations of objects of class `jthread`

Note that the member types `id` and `native_handle_type` are the same for both `std::thread` and `std::jthread` so it does not matter if you use `decltype(mythread)::id` or `std::thread::id`. That way, you can just replace `std::thread` in existing code with `std::jthread` without having to replace anything else.

12.3.1 Using Stop Tokens with std::jthread

Besides the fact that the destructor joins, the major benefit of std::jthread is that it automatically establishes the mechanism to signal a stop. For that, the constructor starting a thread creates a stop source, stores it as a member of the thread object, and passes the corresponding stop token to the called function ***in case*** that function takes an additional stop_token as first parameter.

You can also get the stop source and stop token via member functions from the thread:

```
std::jthread t1{[] (std::stop_token st) {
                  ...
                }};
...
foo(t1.get_token());            // pass stop token to foo()
...
std::stop_source ssrc{t1.get_stop_source()};
ssrc.request_stop();            // request stop on stop token of t1
```

Because get_token() and get_stop_source() return by value, both the stop source and the stop token can even be used after a thread has been detached and runs in the background.

Using Stop Tokens with a Collection of std::jthreads

If you start multiple jthreads, each thread has its own stop token. Note that this might result in a situation that stopping all threads might take longer than expected. Consider the following code:

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
                                  while (!st.stop_requested()) {
                                    ...
                                  }
                                }});
  }
  ...
} // destructor stops all threads
```

At the end of the loop, the destructor stops all running threads similarly to the following code:

```
for (auto& t : threads) {
  t.request_stop();
  t.join();
}
```

This means that we always wait for one thread to end before we signal stop to the next thread.

Code like this can be improved by requesting a stop for all threads before calling `join()` for all of them (via the destructor):

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
                                  while (!st.stop_requested()) {
                                    ...
                                  }
                                }});
  }
  ...
  // BETTER: request stops for all threads before we start to join them:
  for (auto& t : threads) {
    t.request_stop();
  }
} // destructor stops all threads
```

Now we first request a stop for all threads and threads might end before the destructor calls `join()` for the threads to finish. An example of this technique is demonstrated by the destructor of a coroutine thread pool.

Using the Same Stop Token for Multiple `std::jthreads`

It might also be necessary to request a stop for multiple threads using the same stop token. This can be done easily. Just create the stop token yourself or take the stop token from the first thread that has been started and start the (other) threads with this stop token as first argument. For example:

```
// initialize a common stop token for all threads:
std::stop_source allStopSource;
std::stop_token allStopToken{allStopSource.get_token()};

for (int i = 0; i < 9; ++i) {
  threads.push_back(std::jthread{[] (std::stop_token st) {
                                   ...
                                   while (!st.stop_requested()) {
                                     ...
                                   }
                                 },
                                 allStopToken   // pass token to this thread
                                });
}
```

Remember that the callable usually just takes all arguments passed. The internal stop token of the thread started is used only if Only if there is an *additional* stop token parameter for which no argument is passed.

See lib/atomicref.cpp for a complete example.

12.4 Afternotes

The request that threads should join was first proposed by Herb Sutter in `http://wg21.link/n3630` as a fix for `std::thread`. The finally accepted wording was formulated by Nicolai Josuttis, Lewis Baker, Billy O'Neal, Herb Sutter, and Anthony Williams in `http://wg21.link/p0660r10`.

This page is intentionally left blank

Chapter 13

Concurrency Features

After the introduction to `std::jthread` and stop tokens in the previous chapter, this chapter documents all other concurrency features introduced by C++20:

- Latches and barriers
- Counting and binary semaphores
- Various extensions for atomic types
- Synchronized output streams

13.1 Thread Synchronization with Latches and Barriers

Two new types provide new mechanisms for synchronizing asynchronous computation/processing of multiple threads:

- **Latches** allow you to have a one-time synchronization where threads can wait for multiple tasks to finish.
- **Barriers** allow you to have a repeated synchronization of multiple threads where you have to react when they all have done their current/next processing.

13.1.1 Latches

A latch is a new synchronization mechanism for concurrent execution that supports a single-use asynchronous countdown. Starting with an initial integral value, various threads can atomically count this value down to zero. The moment the counter reaches zero, all threads waiting for this countdown continue.

Consider the following example:

lib/latch.cpp

```
#include <iostream>
#include <array>
#include <thread>
#include <latch>
```

```
using namespace std::literals;   // for duration literals

void loopOver(char c) {
  // loop over printing the char c:
  for (int j = 0; j < c/2; ++j) {
    std::cout.put(c).flush();
    std::this_thread::sleep_for(100ms);
  }
}

int main()
{
  std::array tags{'.', '?', '8', '+', '-'};   // tags we have to perform a task for

  // initialize latch to react when all tasks are done:
  std::latch allDone{tags.size()};     // initialize countdown with number of tasks

  // start two threads dealing with every second tag:
  std::jthread t1{[tags, &allDone] {
                    for (unsigned i = 0; i < tags.size(); i += 2) {   // even indexes
                      loopOver(tags[i]);
                      // signal that the task is done:
                      allDone.count_down();   // atomically decrement counter of latch
                    }
                    ...
                  }};
  std::jthread t2{[tags, &allDone] {
                    for (unsigned i = 1; i < tags.size(); i += 2) {   // odd indexes
                      loopOver(tags[i]);
                      // signal that the task is done:
                      allDone.count_down();   // atomically decrement counter of latch
                    }
                    ...
                  }};
  ...
  // wait until all tasks are done:
  std::cout << "\nwaiting until all tasks are done\n";
  allDone.wait();                          // wait until counter of latch is zero
  std::cout << "\nall tasks done\n";      // note: threads might still run
  ...
}
```

In this example, we start two threads (using std::jthread) to perform a couple of tasks that each process a character of the array `tags`. Thus, the size of `tags` defines the number of tasks. The main thread blocks until all tasks are done. For this:

- We initialize a latch with the number of tags/tasks:
  ```
  std::latch allDone{tags.size()};
  ```
- We let each task decrement the counter when it is done:
  ```
  allDone.count_down();
  ```
- We let the main thread wait until all tasks are done (the counter is zero):
  ```
  allDone.wait();
  ```

The output of the program might look as follows:

```
.?
waiting until all tasks are done
.??..??.?..?.??.?..??.?..?.??..?.??.?.?.?.?.8??88??88??8?8?8+8+88++8+8+8+88++
8+8+8+8+88++8+8+88++8+8-+---------------------
all tasks done
```

Note that the main thread should ***not*** assume that when the tasks are done, all threads are done. The threads might still do other things and the system might still clean them up. To wait until all threads are done, you have to call `join()` for both threads.

You can also use latches to synchronize multiple threads at a specific point and then continue. However, note that you can do this only once with each latch. One application of this is to ensure (as well as possible) that multiple threads start their actual work together even when starting and initializing the threads might take some time.[1]

Consider the following example:

lib/latchready.cpp

```cpp
#include <iostream>
#include <array>
#include <vector>
#include <thread>
#include <latch>
using namespace std::literals;   // for duration literals

int main()
{
  std::size_t numThreads = 10;

  // initialize latch to start the threads when all of them have been initialized:
  std::latch allReady = 10;      // initialize countdown with number of threads

```

[1] Thanks to Anthony Williams for describing this scenario.

```
  // start numThreads threads:
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    std::jthread t{[i, &allReady] {
                     // initialize each thread (simulate to take some time):
                     std::this_thread::sleep_for(100ms * i);
                     ...
                     // synchronize threads so that all start together here:
                     allReady.arrive_and_wait();

                     // perform whatever the thread does
                     // (loop printing its index):
                     for (int j = 0; j < i + 5; ++j) {
                       std::cout.put(static_cast<char>('0' + i)).flush();
                       std::this_thread::sleep_for(50ms);
                     }
                   }};
    threads.push_back(std::move(t));
  }
  ...
}
```

We start numThreads threads (using std::jthread), which take their time getting initialized and started. To allow them start their functionality together, we use a latch to block until all threads started have been initialized and started. For this:

- We initialize the latch with the number of threads:

  ```
  std::latch allReady{numThreads};
  ```

- We let each thread decrement the counter to wait until the initialization of all threads is done:

  ```
  allReady.arrive_and_wait();   // count_down() and wait()
  ```

The output of the program might look as follows:

```
86753421098675342019901425376886735241907863524910768352491942538679453876945
876957869786789899
```

You can see that all 10 threads (each printing its index) start more or less together.

Without the latch, the output might look as follows:

```
00101021021321324132435243524365463547635746854768547968579685796587968769876
987987987989898999
```

Here, the threads started early are already running while the latter have not been started yet.

Latches in Detail

The class `std::latch` is declared in the header file `<latch>`. Table *Operations of objects of class* `latch` lists the API of `std::latch`.

Operation	Effect
latch `l{counter}`	Creates a latch with *`counter`* as the starting value for the countdown
`l.count_down()`	Atomically decrements the counter (if not `0` yet)
`l.count_down(val)`	Atomically decrements the counter by *`val`*
`l.wait()`	Blocks until the counter of the latch is `0`
`l.try_wait()`	Yields whether the counter of the latch is `0`
`l.arrive_and_wait()`	Calls `count_down()` and `wait()`
`l.arrive_and_wait(val)`	Calls `count_down(val)` and `wait()`
`max()`	Static function that yields the maximum possible value for *`counter`*

Table 13.1. Operations of objects of class `latch`

Note that you cannot copy or move (assign) a latch.

Note also that passing the size of a container (except `std::array`) as the initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::latch l1{10};                      // OK
std::latch l2{10u};                     // warnings may occur

std::vector<int> coll{...};
...
std::latch l3{coll.size()};             // ERROR
std::latch l4 = coll.size();            // ERROR
std::latch l5(coll.size());             // OK (no narrowing checked)
std::latch l6{int(coll.size())};        // OK
std::latch l7{ssize(coll)};             // OK (see std::ssize())
```

13.1.2 Barriers

A barrier is a new synchronization mechanism for concurrent execution that allows you to synchronize multiple asynchronous tasks multiple times. After setting an initial count, multiple threads can count it down and wait until the counter reaches zero. However, in contrast to latches, when zero is reached, an (optional) callback is called and the counter reinitializes to the initial count again.

A barrier is useful when multiple threads repeatedly compute/perform something together. Whenever all threads have done their task, the optional callback can process the result or new state and after that, the asynchronous computation/processing can continue with the next round.

As an example, consider that we repeatedly use multiple threads to compute the square root of multiple values:

lib/barrier.cpp

```
#include <iostream>
#include <format>
#include <vector>
#include <thread>
#include <cmath>
#include <barrier>

int main()
{
  // initialize and print a collection of floating-point values:
  std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

  // define a lambda function that prints all values
  // - NOTE: has to be noexcept to be used as barrier callback
  auto printValues = [&values] () noexcept{
                       for (auto val : values) {
                         std::cout << std::format(" {:<7.5}", val);
                       }
                       std::cout << '\n';
                     };
  // print initial values:
  printValues();

  // initialize a barrier that prints the values when all threads have done their computations:
  std::barrier allDone{int(values.size()),  // initial value of the counter
                       printValues};        // callback to call whenever the counter is 0

  // initialize a thread for each value to compute its square root in a loop:
  std::vector<std::jthread> threads;
  for (std::size_t idx = 0; idx < values.size(); ++idx) {
    threads.push_back(std::jthread{[idx, &values, &allDone] {
                                     // repeatedly:
                                     for (int i = 0; i < 5; ++i) {
                                       // compute square root:
                                       values[idx] = std::sqrt(values[idx]);
                                       // and synchronize with other threads to print values:
                                       allDone.arrive_and_wait();
                                     }
                                   }});
  }
  ...
}
```

After declaring an array of floating-point values, we define a function to print them out (using `std::format()` for formatted output):

```
// initialize and print a collection of floating-point values:
std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

// define a lambda function that prints all values
// - NOTE: has to be noexcept to be used as barrier callback
auto printValues = [&values] () noexcept{
                     for (auto val : values) {
                       std::cout << std::format(" {:<7.5}", val);
                     }
                     std::cout << '\n';
                   };
```

Note that the callback has to be declared with `noexcept`.

Our goal is to use multiple threads so that each thread deals with one value. In this case, we repeatedly compute the square roots of the values. For this:

- We initialize barrier `allDone` to print all values whenever all threads have done their next computation: the number of tags/tasks:

  ```
  std::barrier allDone{int(values.size()),  // initial value of the counter
                       printValues};         // callback to call whenever the counter is 0
  ```

  Please note that the constructor should take a signed integral value. Otherwise, the code might not compile.
- In the loop, each thread decrements the counter after its computation and waits until the counter reaches zero (i.e., all other threads also have signaled that they are done):

  ```
  ...
  allDone.arrive_and_wait();
  ```
- When the counter reaches zero, the callback is called to print the result.

  Note that the callback is called by the thread that finally decremented the counter to zero. This means that in the loop, the callback is called by different threads.

The output of the program might look as follows:

```
1       2       3       4       5       6       7       8
1       1.4142  1.7321  2       2.2361  2.4495  2.6458  2.8284
1       1.1892  1.3161  1.4142  1.4953  1.5651  1.6266  1.6818
1       1.0905  1.1472  1.1892  1.2228  1.251   1.2754  1.2968
1       1.0443  1.0711  1.0905  1.1058  1.1185  1.1293  1.1388
1       1.0219  1.0349  1.0443  1.0516  1.0576  1.0627  1.0671
```

The API of barriers also provide a function to drop threads from this mechanism. This is something you need, for example, to avoid deadlocks when the loop runs until a stop has been signaled.

The code for the threads started might then look as follows:

```
// initialize a thread for each value to compute its square root in a loop:
std::vector<std::jthread> threads;
for (std::size_t idx = 0; idx < values.size(); ++idx) {
  threads.push_back(std::jthread{[idx, &values, &allDone] (std::stop_token st) {
                                   // repeatedly:
                                   while (!st.stop_requested()) {
                                     // compute square root:
                                     values[idx] = std::sqrt(values[idx]);
                                     // and synchronize with other threads to print values:
                                     allDone.arrive_and_wait();
                                   }
                                   // drop thread from barrier so that other threads not wait:
                                   allDone.arrive_and_drop();
                                 }});
}
```

The lambda called by each thread now takes a stop token to react when a stop has been requested (explicitly or by calling the destructor for the thread). If the main thread signals the threads to stop the computing (e.g., by calling `threads.clear()`), it is important that each thread drops itself from the barrier:

```
allDone.arrive_and_drop();
```

This call counts down the counter ***and*** ensures that the next initial value for the value is decremented. In the next round (the other threads might still be running) the barrier will no longer wait for the dropped thread.

See *lib/barrierstop.cpp* for a complete example.

Barriers in Detail

The class `std::barrier` is declared in the header file `<barrier>`. Table *Operations of objects of class* `barrier` lists the API of `std::barrier`.

Operation	Effect
barrier *b*{*num*}	Creates a barrier for *num* asynchronous tasks
barrier *b*{*num*, *cb*}	Creates a barrier for *num* asynchronous tasks and *cb* as callback
b.arrive()	Marks one task as done and yields an arrival token
b.arrive(*val*)	Marks *val* tasks as done and yields an arrival token
b.wait(*arrivalToken*)	Blocks until all tasks have been done and the callback has been called (if there is one)
b.**arrive_and_wait()**	Marks one task as done and blocks until all tasks have been done and the callback has been called (if there is one)
b.**arrive_and_drop()**	Marks one task as done and decrements the number of tasks to repeatedly perform
max()	Static function that yields the maximum possible value for *num*

Table 13.2. Operations of objects of class `barrier`

`std::barrier<>` is a class template with the type of the callback as a template parameter. Usually, the type is deduced by *class template argument deduction*:

```
void callback() noexcept;   // forward declaration
...

std::barrier b{6, callback};   // deduces std::barrier<decltype(callback)>
```

Note that the C++ standard requires that callbacks for barriers guarantee not to throw. Therefore, to be portable, you have to declare the function or lambda with `noexcept`. If no callback is passed, an implementation-specific type is used, representing an operation that has no effects.

The call

```
l.arrive_and_wait();
```

is equivalent to

```
l.wait(l.arrive());
```

This means that the `arrive()` function returns an *arrival token* of type `std::barrier::arrival_token`, which ensures that the barrier knows which of the threads to wait for. Otherwise, it cannot handle `arrive_and_drop()` correctly.

Note that you cannot copy or move (assign) a barrier.

Note also that passing the size of a container (except `std::array`) as an initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::barrier b1{10, cb};                // OK
std::barrier b2{10u, cb};               // warnings may occur

std::vector<int> coll{...};
...
std::barrier b3{coll.size(), cb};       // ERROR
std::barrier b4(coll.size(), cb);       // OK (no narrowing checked)
std::barrier b5{int(coll.size()), cb};  // OK
std::barrier b6{std::ssize(coll), cb};  // OK (see std::ssize())
```

The function `std::ssize()` was introduced in C++20.

13.2 Semaphores

C++20 introduced new types for dealing with semaphores. Semaphores are lightweight synchronization primitives that allow you to synchronize or restrict access to one or a group of resources.

You can use them like mutexes, with the benefit that the threads granting access to a resource do not necessarily have to be the same threads that acquired the access to the resource. You can also use them to limit the availability of resources, such as enabling and disabling the use of threads in a thread pool.

There are two semaphore types provided by the C++ standard library:

- `std::`**`counting_semaphore<>`** to limit the use of multiple resources up to a maximum value
- `std::`**`binary_semaphore<>`** to limit the use of a single resource

13.2.1 Example of Using Counting Semaphores

The following program demonstrates how semaphores operate in general:

lib/semaphore.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>
using namespace std::literals;   // for duration literals

int main()
{
  std::queue<char> values;   // queue of values
  std::mutex valuesMx;       // mutex to protect access to the queue

  // initialize a queue with multiple sequences from 'a' to 'z':
  // - no mutex because no other thread is running yet
  for (int i = 0; i < 1000; ++i) {
    values.push(static_cast<char>('a' + (i % ('z'-'a'))));
  }

  // create a pool of numThreads threads:
  // - limit their availability with a semaphore (initially none available):
  constexpr int numThreads = 10;
  std::counting_semaphore<numThreads> enabled{0};

  // create and start all threads of the pool:
  std::vector<std::jthread> pool;
  for (int idx = 0; idx < numThreads; ++idx) {
    std::jthread t{[idx, &enabled, &values, &valuesMx] (std::stop_token st) {
                      while (!st.stop_requested()) {
                        // request thread to become one of the enabled threads:
                        enabled.acquire();

                        // get next value from the queue:
                        char val;
                        {
                          std::lock_guard lg{valuesMx};
                          val = values.front();
                          values.pop();
                        }
```

```
                              // print the value 10 times:
                              for (int i = 0; i < 10; ++i) {
                                std::cout.put(val).flush();
                                auto dur = 130ms * ((idx % 3) + 1);
                                std::this_thread::sleep_for(dur);
                              }

                              // remove thread from the set of enabled threads:
                              enabled.release();
                            }
                          }};
    pool.push_back(std::move(t));
  }

  std::cout << "== wait 2 seconds (no thread enabled)\n" << std::flush;
  std::this_thread::sleep_for(2s);

  // enable 3 concurrent threads:
  std::cout << "== enable 3 parallel threads\n" << std::flush;
  enabled.release(3);
  std::this_thread::sleep_for(2s);

  // enable 2 more concurrent threads:
  std::cout << "\n== enable 2 more parallel threads\n" << std::flush;
  enabled.release(2);
  std::this_thread::sleep_for(2s);

  // Normally we would run forever, but let's end the program here:
  std::cout << "\n== stop processing\n" << std::flush;
  for (auto& t : pool) {
    t.request_stop();
  }
}
```

In this program, we start 10 threads but limit how many of them are allowed to actively run and process data. To do this, we initialize a semaphore with the maximum possible number we might allow (10) and the initial number of resources allowed (zero):

```
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> enabled{0};
```

You might wonder why we have to specify a maximum as a template parameter. The reason is that with this compile-time value, the library can decide to switch to the most efficient implementation possible (native support might be possible only up to a certain value or if the maximum is 1, we may be able to use simplifications).

In each thread, we use the semaphore to ask for permission to run. We try to "acquire" one of the available resources to start the task and "release" the resource for other use when we are done:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 enabled.acquire();
                 ...

                 // remove thread from the set of enabled threads:
                 enabled.release();
               }
             }}
```

Initially, we are blocked, because the semaphore was initialized with zero so that no resources are available. Later, however, we use the semaphore to allow three threads to act concurrently:

```
// enable 3 concurrent threads:
enabled.release(3);
```

And even later, we allow two more threads to run concurrently:

```
// enable 2 more concurrent threads:
enabled.release(2);
```

If you want to sleep or do something else if you cannot get a resource, you can use `try_acquire()`:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 if (enabled.try_acquire()) {
                   ...

                   // remove thread from the set of enabled threads:
                   enabled.release();
                 }
                 else {
                   ...
                 }
               }
             }}
```

You can also try to acquire a resource for a limited time using `try_acquire_for()` or `try_acquire_until()`:

```
std::jthread{[&, idx] (std::stop_token st) {
               while (!st.stop_requested()) {
                 // request thread to become one of the enabled threads:
                 if (enabled.try_acquire_for(100ms)) {
                   ...
```

```
                  // remove thread from the set of enabled threads:
                  enabled.release();
                }
              }
            }}
```

This would allow us to double check the status of the stop token from time to time.

Thread Scheduling Is Not Fair

Note that threads are not necessarily scheduled fairly. Therefore, the main thread might need some time (or even forever) to become scheduled when it wants to end the program. It is also a good approach to request all threads to stop before we call the destructors of the threads. Otherwise, we request the stops significantly later when the previous threads have finished.

Another consequence of the fact that threads are not scheduled fairly is that there is no guarantee that threads that wait the longest time are preferred. Usually, the contrary is true: if a thread scheduler already has a thread running calling `release()` and it immediately calls `acquire()`, the scheduler keeps the thread running ("great, I do not need a context switch"). Therefore, we have no guarantee which one of multiple threads that wait due to calling `acquire()` are woken up. It might be the case that the same thread(s) is/are always used.

As a consequence, you should ***not*** take the next value out of the queue before you ask for permission to run.

```
// BAD if order of processing matters:
{
  std::lock_guard lg{valuesMx};
  val = values.front();
  values.pop();
}
enabled.acquire();
...
```

It might be the case that the value `val` is processed after several other values read later or even never processed. Ask for permission before you read the next value:

```
enabled.acquire();
{
  std::lock_guard lg{valuesMx};
  val = values.front();
  values.pop();
}
...
```

For the same reason, you cannot easily reduce the number of enabled threads. Yes, you can try to call:

```
// reduce the number of enabled concurrent threads by one:
enabled.acquire();
```

However, we do not know when this statement is processed. As threads are not treated fairly, the reaction to reduce the number of enabled threads may take very long or even take forever.

For a fair way to deal with queues and an immediate reaction to resource limitations, you might want to use the new `wait()` and notify mechanism of atomics.

13.2.2 Example of Using Binary Semaphores

For semaphores, a special type `std::binary_semaphore` has been defined, which is just a shortcut for `std::counting_semaphore<1>` so that it can only enable or disable the use of a single resource.

You could also use it as a mutex, with the benefit that the thread releasing the resource does not have to be the same thread that formerly acquired the resource. However, a more typical application is a mechanism to signal/notify a thread from another thread. In contrast to condition variables, you can do that multiple times.

Consider the following example:

lib/semaphorenotify.cpp

```
#include <iostream>
#include <chrono>
#include <thread>
#include <semaphore>
using namespace std::literals;   // for duration literals

int main()
{
  int sharedData;
  std::binary_semaphore dataReady{0};  // signal there is data to process
  std::binary_semaphore dataDone{0};   // signal processing is done

  // start threads to read and process values by value:
  std::jthread process{[&] (std::stop_token st) {
                         while(!st.stop_requested()) {
                           // wait until the next value is ready:
                           // - timeout after 1s to check stop_token
                           if (dataReady.try_acquire_for(1s)) {
                             int data = sharedData;

                             // process it:
                             std::cout << "[process] read " << data << std::endl;
                             std::this_thread::sleep_for(data * 0.5s);
                             std::cout << "[process]      done" << std::endl;

                             // signal processing done:
                             dataDone.release();
                           }
```

```
                            else {
                              std::cout << "[process] timeout" << std::endl;
                            }
                          }
                        }};

  // generate a couple of values:
  for (int i = 0; i < 10; ++i) {
    // store next value:
    std::cout << "[main] store " << i << std::endl;
    sharedData = i;

    // signal to start processing:
    dataReady.release();

    // wait until processing is done:
    dataDone.acquire();
    std::cout << "[main] processing done\n" << std::endl;
  }
  // end of loop signals stop
}
```

We use two binary semaphores to let one thread notify another thread:

- With dataReady, the main thread notifies the thread process that there is new data to process in sharedData.
- With dataDone, the processing thread notifies the main thread that the data was processed.

Both semaphores are initialized by zero so that by default, the acquiring thread blocks. The moment the notifying thread calls release(), the acquiring thread unblocks so that it can react.

The output of the program is something like the following:

```
[main] store 0
[process] read 0
[process]      done
[main] processing done

[main] store 1
[process] read 1
[process]      done
[main] processing done

[main] store 2
[process] read 2
[process]      done
[main] processing done
```

```
[main] store 3
[process] read 3
[process]      done
[main] processing done
...

[main] store 9
[process] read 9
[process]      done
[main] processing done

[process] timeout
```

Note that the processing thread only acquires for a limited time using `try_acquire_for()`. The return value yields whether it was notified (access to the resource). This allows the thread to check from time to time whether a stop was signaled (as is done after the main thread ends).

Another application of binary semaphores is to wait for the end of a coroutine running in a different thread.

Semaphores in Detail

The class template `std::counting_semaphore<>` is declared in the header file `<semaphore>` together with the shortcut `std::binary_semaphore` for `std::counting_semaphore<1>`:

```
namespace std {
  template<ptrdiff_t least_max_value = implementation-defined >
  class counting_semaphore;

  using binary_semaphore = counting_semaphore<1>;
}
```

Table *Operations of objects of class* `counting_semaphore<>` *and* `binary_semaphore` lists the API of semaphores.

Note that you cannot copy or move (assign) a semaphore.

Note also that passing the size of a container (except `std::array`) as an initial value of the counter is an error. The constructor takes a `std::ptrdiff_t`, which is signed, so that you get the following behavior:

```
std::counting_semaphore s1{10};                  // OK
std::counting_semaphore s2{10u};                 // warnings may occur

std::vector<int> coll{...};
...
std::counting_semaphore s3{coll.size()};         // ERROR
std::counting_semaphore s4 = coll.size();        // ERROR
std::counting_semaphore s4(coll.size());         // OK (no narrowing checked)
std::counting_semaphore s6{int(coll.size())};    // OK
std::counting_semaphore s7{std::ssize(coll)};    // OK (see std::ssize())
```

The function `std::ssize()` was introduced in C++20.

Operation	**Effect**
semaphore *s*{*num*}	Creates a semaphore with the counter initialized with *num*
s.**acquire()**	Blocks until it can atomically decrement the counter (requesting one more resource)
s.try_acquire()	Tries to immediately atomically decrement the counter (requesting one more resource) and yields `true` if this was successful
s.try_acquire_for(*dur*)	Tries for duration *dur* to atomically decrement the counter (requesting one more resource) and yields `true` if this was successful
s.try_acquire_until(*tp*)	Tries until timepoint *tp* to atomically decrement the counter (requesting one more resource) and yields `true` if this was successful
s.**release()**	Atomically increments the counter (enabling one more resource)
s.release(*num*)	Atomically adds *num* to the counter (enabling *num* more resource)
max()	Static function that yields the maximum possible value of the counter

Table 13.3. Operations of objects of class `counting_semaphore<>` *and* `binary_semaphore`

13.3 Extensions for Atomic Types

C++20 introduces a couple of new atomic types (dealing with references and shared pointers) and new features for atomic types. In addition, we now have a type `std::atomic<char8_t>`.

13.3.1 Atomic References with `std::atomic_ref<>`

Since C++11, the C++ standard library provides the class template `std::atomic<>` to provide an atomic API for trivially copyable types.

C++20 now introduces the class template `std::atomic_ref<>` to provide an atomic API for trivially copyable ***reference*** types. This allows you to provide a temporary atomic API to an existing object that is usually not atomic. One application would be to initialize an object without caring about concurrency and later use it with different threads.

The reason for giving the type a unique name `atomic_ref` and not just providing `std::atomic<>` for reference types is that users should see that the object might provide non-atomic access and that there are weaker guarantees than for `std::atomic<>`.

An Example Using Atomic References

The following program demonstrates how to use atomic references:

lib/atomicref.cpp

```
#include <iostream>
#include <array>
#include <algorithm>    // for std::fill_n()
#include <vector>
#include <format>
```

```
#include <random>
#include <thread>
#include <atomic>       // for std::atomic_ref<>
using namespace std::literals;   // for duration literals

int main()
{
  // create and initialize an array of integers with the value 100:
  std::array<int, 1000> values;
  std::fill_n(values.begin(), values.size(), 100);

  // initialize a common stop token for all threads:
  std::stop_source allStopSource;
  std::stop_token allStopToken{allStopSource.get_token()};

  // start multiple threads concurrently decrementing the value:
  std::vector<std::jthread> threads;
  for (int i = 0; i < 9; ++i) {
    threads.push_back(std::jthread{
      [&values] (std::stop_token st) {
        // initialize random engine to generate an index:
        std::mt19937 eng{std::random_device{}()};
        std::uniform_int_distribution distr{0, int(values.size()-1)};

        while (!st.stop_requested()) {
          // compute the next index:
          int idx = distr(eng);

          // enable atomic access to the value with the index:
          std::atomic_ref val{values[idx]};

          // and use it:
          --val;
          if (val <= 0) {
            std::cout << std::format("index {} is zero\n", idx);
          }
        }
      },
      allStopToken   // pass the common stop token
    });
  }

  // after a while/event request to stop all threads:
  std::this_thread::sleep_for(0.5s);
  std::cout << "\nSTOP\n";
  allStopSource.request_stop();
  ...
}
```

We first create and initialize an array of 1000 integral values without using atomics:

```
std::array<int, 1000> values;
std::fill_n(values.begin(), values.size(), 100);
```

Later, however, we start multiple threads that concurrently decrement these values. The point is that only in that context do we use the values as atomic integers. For this, we initialize a `std::atomic_ref<>`:

```
std::atomic_ref val{values[idx]};
```

Note that due to class template argument deduction, we do not have to specify the type of the object we refer to.

The effect of this initialization is that the access to the value happens atomically, when using the atomic reference:

- `--val` decrements the value atomically
- `val <= 0` loads the value atomically to compare it with `0` (the expression uses an implicit type conversion to the underlying type after reading the value)

You might consider implementing the latter as `val.load() <= 0` to document that an atomic interface is used.

Note that the different threads of this program do ***not*** use the same `atomic_ref<>` objects. This is fine. `std::atomic_ref<>` guarantees that all concurrent access to a specific object through *any* `atomic_ref<>` created for it is synchronized.[2]

Features of Atomic References

The header file for atomic references is also `<atomic>`. As for `std::atomic<>` specializations for raw pointers, integral types, and (since C++20) floating-point types exist:

```
namespace std {
  template<typename T> struct atomic_ref;        // primary template

  template<typename T> struct atomic_ref<T*>;    // partial specialization for pointers

  template<> struct atomic_ref<integralType>;    // full specializations for integral types

  template<> struct atomic_ref<floatType>;       // full specializations for floating-point types
}
```

[2] It is up to the implementations of the standard library how to ensure this. If a mutex or lock is necessary, a global hash table might be used, where for each address of the wrapped objects an associated lock is stored. This is no different to the way implementations may implement `std::atomic<>` for user-defined types.

Atomic reference have the following restrictions compared to `std::atomic<>`:

- There is no `volatile` support.
- The objects to be referred to might need an alignment that is greater than the usual alignment for the underlying type. The static member `std::atomic_ref<`*`type`*`>::required_alignment` provides this minimum alignment.

Atomic references have the following extensions compared to `std::atomic<>`:

- A copy constructor to create another reference to the same underlying object (but an assignment operator is provided only to assign the underlying values).
- Constness is ***not*** propagated to the wrapped object. This means that you can assign a new value to a `const std::atomic_ref<>`:

  ```
  MyType x, y;
  const std::atomic_ref cr{x};
  cr = y;      // OK (would not be OK for const std::atomic<>)
  ```

- Thread synchronization support with `wait()`, `notify_one()`, and `notify_all()`, as is now provided for all atomic types.

Regarding all other aspects, the same features as for `std::atomic<>` are provided:

- The type provides both a high-level API with memory barriers and a low-level API to disable them.
- The static member `is_always_lock_free` and the non-static member function `is_lock_free()` yield whether atomic support is lock-free. This might still depend on the alignment used.

Atomics References in Detail

Atomic references provide the same API as the corresponding atomic types. For a trivially copyable type, pointer type, integral type, or floating-point type ***T***, the C++ standard library provides `std::atomic_ref<`***T***`>` with the same atomic API as `std::atomic<`***T***`>`. The atomic reference types are also provided in the header file `<atomic>`.

This also means that with the static member `is_always_lock_free()` or the non-static member function `is_lock_free()`, you can check whether an atomic type uses locks internally to be atomic. If not, you have native hardware support for the atomic operations (which is a prerequisite for using atomics in signal handlers). The check for a type looks as follows:

```
if constexpr(std::atomic<int>::is_always_lock_free) {
  ...
}
else {
  ...     // special handling if locks are used
}
```

The check for a specific object (whether it is lock-free might depend on the alignment) looks as follows:

```
std::atomic_ref val{values[idx]};
if (val.is_lock_free()) {
  ...
}
else {
  ...     // special handling if locks are used
}
```

Note that for a type T, std::atomic_ref<T> might ***not*** be lock-free although type std::atomic<T> is lock-free.

An object referenced by an atomic reference might also have to satisfy possibly architecture-specific constraints. For example, the object might need to be properly aligned in memory or might not be allowed to reside in GPU register memory. For the required alignment there is a static member:

```
namespace std {
  template<typename T> struct atomic_ref {
    static constexpr size_t required_alignment;
    ...
  };
}
```

The value is at least alignof(T). However, the value might be, for example, 2*alignof(double) to support lock-free operations for std::complex<double>.

13.3.2 Atomic Shared Pointers

C++11 introduced shared pointers with an optional atomic interface. With functions like atomic_load(), atomic_store(), and atomic_exchange(), you could access the values that shared pointers refer to concurrently. However, the problem was that you could still use these shared pointers with the non-atomic interface, which would then undermine all atomic use of the shared pointers.

C++20 now provides partial specializations for shared pointers and weak pointers:

- std::atomic<std::shared_ptr<T>>
- std::atomic<std::weak_ptr<T>>

The former atomic API for shared pointers is now deprecated.

Note that atomic shared/weak pointers do ***not*** provide the additional operations that atomic raw pointers provide. They provide the same API as std::atomic<> provides in general for a trivially copyable type T.

Thread synchronization support with wait(), notify()_one(), and notify_all() is provided. The low-level atomic interface with the option to use the parameters for memory order is also supported.

Example of Using Atomic Shared Pointers

The following example demonstrates the use of an atomic shared pointer that is used as head of a linked list of shared values.

lib/atomicshared.cpp

```
#include <iostream>
#include <thread>
#include <memory>                // includes <atomic> now
using namespace std::literals;   // for duration literals

template<typename T>
class AtomicList {
 private:
  struct Node {
    T val;
    std::shared_ptr<Node> next;
  };
  std::atomic<std::shared_ptr<Node>> head;
 public:
  AtomicList() = default;

  void insert(T v) {
    auto p = std::make_shared<Node>();
    p->val = v;
    p->next = head;
    while (!head.compare_exchange_weak(p->next, p)) {   // atomic update
    }
  }

  void print() const {
    std::cout << "HEAD";
    for (auto p = head.load(); p; p = p->next) {   // atomic read
      std::cout << "->" << p->val;
    }
    std::cout << std::endl;
  }
};

int main()
{
  AtomicList<std::string> alist;

  // populate list with elements from 10 threads:
  {
    std::vector<std::jthread> threads;
    for (int i = 0; i < 100; ++i) {
      threads.push_back(std::jthread{[&, i]{
                                       for (auto s : {"hi", "hey", "ho", "last"}) {
                                         alist.insert(std::to_string(i) + s);
                                         std::this_thread::sleep_for(5ns);
                                       }
                                     }});
    }
  } // wait for all threads to finish

  alist.print();     // print resulting list
}
```

The program may have the following output:

```
HEAD->94last->94ho->76last->68last->57last->57ho->60last->72last->... ->1hey->1hi
```

As usual with atomic operations, you could also write:

```
for (auto p = head.load(); p; p = p->next)
```

instead of:

```
for (auto p = head.load(); p.load(); p = p->next)
```

Example of Using Atomic Weak Pointers

The following example demonstrates the use of an atomic weak pointer:

lib/atomicweak.cpp

```
#include <iostream>
#include <thread>
#include <memory>                  // includes <atomic> now
using namespace std::literals;     // for duration literals


int main()
{
  std::atomic<std::weak_ptr<int>> pShared;   // pointer to current shared value (if one exists)

  // loop to set shared value for some time:
  std::atomic<bool> done{false};
  std::jthread updates{[&] {
                         for (int i = 0; i < 10; ++i) {
                           {
                             auto sp = std::make_shared<int>(i);
                             pShared.store(sp);     // atomic update
                             std::this_thread::sleep_for(0.1s);
                           }
                           std::this_thread::sleep_for(0.1s);
                         }
                         done.store(true);
                       }};

  // loop to print shared value (if any):
  while (!done.load()) {
    if (auto sp = pShared.load().lock()) {   // atomic read
      std::cout << "shared: " << *sp << '\n';
    }
    else {
      std::cout << "shared: <no data>\n";
    }
    std::this_thread::sleep_for(0.07s);
  }
}
```

Note that in this program, we do not have to make the shared pointer atomic because it is used by only one thread. The only issue is that two threads concurrently update or use the weak pointer.

The program may have the following output:

```
shared: <no data>
shared: 0
shared: <no data>
shared: 1
shared: <no data>
shared: <no data>
shared: 2
shared: <no data>
shared: <no data>
shared: 3
shared: <no data>
shared: 4
shared: 4
shared: <no data>
shared: 5
shared: 5
shared: <no data>
shared: 6
shared: <no data>
shared: <no data>
shared: 7
shared: <no data>
shared: 8
shared: 8
shared: <no data>
shared: 9
shared: 9
shared: <no data>
```

As usual with atomic operations, you could also write:

```
pShared = sp;
```

instead of:

```
pShared.store(sp);
```

13.3.3 Atomic Floating-Point Types

Both `std::atomic<>` and `std::atomic_ref<>` now provide full specializations for types `float`, `double`, and `long double`.

In contrast to the primary template for arbitrary trivially copyable types, they provide the additional atomic operations to add and subtract a value:[3]

- `fetch_add()`, `fetch_sub()`
- `operator+=`, `operator-=`

Thus, the following is possible now:

```
std::atomic<double> d{0};
...
d += 10.3;     // OK since C++20
```

13.3.4 Thread Synchronization with Atomic Types

All atomic types (`std::atomic<>`, `std::atomic_ref<>`, and `std::atomic_flag`) now provide a simple API to let threads block and wait for changes of their values caused by other threads.

Thus, for an atomic value:

```
std::atomic<int> aVal{100};
```

or an atomic reference:

```
int value = 100;
std::atomic_ref<int> aVal{value};
```

you can define that you want to wait until the referenced value has changed:

```
int lastValue = aVal.load();
aVal.wait(lastValue);  // block unless/until value changed (and notified)
```

If the value of the referenced object does not match the passed argument, it returns immediately. Otherwise, it blocks until `notify_one()` or `notify_all()` has been called for the atomic value or reference:

```
--aVal;                // atomically modify the (referenced) value
aVal.notify_all();     // notify all threads waiting for a change
```

However, as for condition variables, `wait()` might end due to a spurious wake-up (so ***without*** a called notification). Therefore, you should always double check the value after the `wait()`.

The code to wait for a specific atomic value might look as follows:

```
while ((int val = aVal.load()) != expectedVal) {
  aVal.wait(val);
  // here, aVal may or may not have changed
}
```

Note that there is no guarantee that you will get all updates. Consider the following program:

[3] In contrast to specializations to integral types, which also provide atomic support to increment/decrement values and perform bit-wise modifications.

lib/atomicwait.cpp

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std::literals;

int main()
{
  std::atomic<int> aVal{0};

  // reader:
  std::jthread tRead{[&] {
                       int lastX = aVal.load();
                       while (lastX >= 0) {
                         aVal.wait(lastX);
                         std::cout << "=> x changed to " << lastX << std::endl;
                         lastX = aVal.load();
                       }
                       std::cout << "READER DONE" << std::endl;
                    }};

  // writer:
  std::jthread tWrite{[&] {
                        for (int newVal : { 17, 34, 3, 42, -1}) {
                          std::this_thread::sleep_for(5ns);
                          aVal = newVal;
                          aVal.notify_all();
                        }
                     }};
  ...
}
```

The output might be:

```
=> x changed to 17
=> x changed to 34
=> x changed to 3
=> x changed to 42
=> x changed to -1
READER DONE
```

or:

```
=> x changed to 17
=> x changed to 3
=> x changed to -1
READER DONE
```

or just:

```
READER DONE
```

Note that the notification functions are `const` member functions.

Fair Ticketing with Atomic Notifications

One application of using using atomic `wait()` and notifications is to use them like mutexes. This often pays off because using mutexes might be significantly more expensive.

Here is a example where we use atomics to implement a fair processing of values in a queue (compare with the unfair version using semaphores). Although multiple threads might wait, only a limited number of them might run. And by using a ticketing system, we ensure that the elements in the queue are processed in order:[4]

lib/atomicticket.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <atomic>
#include <semaphore>
using namespace std::literals;      // for duration literals

int main()
{
  char actChar = 'a';               // character value iterating endless from 'a' to 'z'
  std::mutex actCharMx;             // mutex to access actChar

  // limit the availability of threads with a ticket system:
  std::atomic<int> maxTicket{0};    // maximum requested ticket no
  std::atomic<int> actTicket{0};    // current allowed ticket no

  // create and start a pool of numThreads threads:
  constexpr int numThreads = 10;
  std::vector<std::jthread> threads;
  for (int idx = 0; idx < numThreads; ++idx) {
    threads.push_back(std::jthread{[&, idx] (std::stop_token st) {
                                   while (!st.stop_requested()) {
                                     // get next character value:
                                     char val;
                                     {
```

[4] The idea for this example is based on an example by Bryce Adelstein Lelbach in his talk *The C++20 Synchronization Library* at the CppCon 2029 (see `http://youtu.be/Zcqwb3CWqs4?t=1810`).

```
                                       std::lock_guard lg{actCharMx};
                                       val = actChar++;
                                       if (actChar > 'z') actChar = 'a';
                                     }

                                     // request a ticket to process it and wait until enabled:
                                     int myTicket{++maxTicket};
                                     int act = actTicket.load();
                                     while (act < myTicket) {
                                       actTicket.wait(act);
                                       act = actTicket.load();
                                     }

                                     // print the character value 10 times:
                                     for (int i = 0; i < 10; ++i) {
                                       std::cout.put(val).flush();
                                       auto dur = 20ms * ((idx % 3) + 1);
                                       std::this_thread::sleep_for(dur);
                                     }

                                     // done, so enable next ticket:
                                     ++actTicket;
                                     actTicket.notify_all();
                                   }
                                 }});
  }

  // enable and disable threads in the thread pool:
  auto adjust = [&, oldNum = 0] (int newNum) mutable {
                  actTicket += newNum - oldNum;             // enable/disable tickets
                  if (newNum > 0) actTicket.notify_all();   // wake up waiting threads
                  oldNum = newNum;
                };

  for (int num : {0, 3, 5, 2, 0, 1}) {
    std::cout << "\n====== enable " << num << " threads" << std::endl;
    adjust(num);
    std::this_thread::sleep_for(2s);
  }

  for (auto& t : threads) {    // request all threads to stop (join done when leaving scope)
    t.request_stop();
  }
}
```

Each thread asks for the next character value and then requests a ticket to process this value:

```
int myTicket{++maxTicket};
```

The thread then waits until the ticket is enabled:

```
int act = actTicket.load();
while (act < myTicket) {
  actTicket.wait(act);
  act = actTicket.load();
}
```

Each time the thread wakes up, it double checks whether its ticket was enabled (`actTicket` is at least `myTicket`). New tickets are enabled by increasing the value of `actTicket` and notifying all waiting threads. This happens either when a thread is done with a processing:

```
++actTicket;
actTicket.notify_all();
```

or when a new number of tickets is enabled:

```
actTicket += newNum - oldNum;               // enable/disable tickets
if (newNum > 0) actTicket.notify_all();     // wake up waiting threads
```

The example might have the following output:

```
====== enable 0 threads

====== enable 3 threads
acbabacabacbaacbabacbacdbdbdcdbdcdecdeddcedegfgegfegegfgegfgegfegfhhfhfhhfhfhijhjjhijhji
jkjkijkjkijkkliklkilkkilmmlimlmilmmlnmlmnml
====== enable o5 threads
pmpnoqrpropnqpronpqorppnorqppornqsrosnqrsosnrqosrsntqstustvqstuvstqtvwuwtvxwtxuvwtxwtxvu
wyxwvyxuwvxwyxuvwyxzxvuyzabyuzbabyzubyabzybczabyzbcabdzbdcazdeedczadedecfafdefcedaefdedf
caefgfecghfigfichfgijhcjgijhgkijjkgihjkgjihjkgij
====== enable khg2 threads
ijkihkhkhkkllmllmlmlllmllmnnmnnmnnmnnmnmnoopoopoopoopoopqpqqpqpqpqpqqrrqrqrrsrsrrsrsrsts
tsts
====== enable 0 threads
ststttttt
====== enable 1 threads
uuuuuuuuuvvvvvvvvvvwwwwwwwwwwxxxxxxxxxxyyyyyyyyyzzzzzzzzzzaaaaaaaaaabbbbbbbbbcccccccc
ccddddddddddeeeeeeeeeefffffffffgggggggggghhhhhhhhh
```

Use synchronized output streams to ensure that the output has no interleaved characters.

13.3.5 Extensions for `std::atomic_flag`

Before C++20, there was no way to check the value of a `std::atomic_flag` without setting it, so C++20 added global and member functions to check for the current value:

- `atomic_flag_test(const atomic_flag*) noexcept;`
- `atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;`
- `atomic_flag::test()` const noexcept;
- `atomic_flag::test(memory_order)` const noexcept;

13.4 Synchronized Output Streams

C++20 provides a new mechanism for synchronizing concurrent output to streams.

13.4.1 Motivation for Synchronized Output Streams

If multiple threads write concurrently to a stream, the output usually has to be synchronized:

- In general, concurrent output to a stream causes undefined behavior (it is a *data race*, the C++ term for a race condition with undefined behavior).
- Concurrent output to standard streams such as `std::cout` is supported, but the result is not very useful because characters from different threads might by mixed in any arbitrary order.

For example, consider the following program:

lib/concstream.cpp

```cpp
#include <iostream>
#include <cmath>
#include <thread>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::cout << "squareroot of " << i << " is "
              << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

Three threads concurrently write to `std::cout`. This is valid because a standard output stream is used. However, the output may look like this:

```
squareroot of squareroot of 0 is 0 is 0
0squareroot of squareroot of
01squareroot of  is  is 101 is

1squareroot of squareroot of
12squareroot of  is  is 21 is 1.41421

1.41421squareroot of squareroot of
23squareroot of  is  is 31.41421 is 1.73205
```

```
1.73205squareroot of squareroot of
34squareroot of  is  is 41.73205 is 2

2squareroot of
4 is 2
```

13.4.2 Using Synchronized Output Streams

By using synchronized output streams, we can now synchronize the concurrent output of multiple threads to the same stream. We only have to use a `std::`**`osyncstream`** initialized with the corresponding output stream. For example:

lib/syncstream.cpp

```cpp
#include <iostream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::osyncstream coutSync{std::cout};
    coutSync << "squareroot of " << i << " is "
             << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

Here, the synchronized output buffer synchronizes the output with other output to a synchronized output buffer so that it is flushed only, when the destructor of the synchronized output buffer is called.

As a result, the output looks like this:

```
squareroot of 0 is 0
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 4 is 2
```

The three threads now write line by line to `std::cout`. However, the exact order of the lines written is still open. You can get the same result by implementing the loop as follows:

```
for (int i = 0; i < num ; ++i) {
  std::osyncstream{std::cout} << "squareroot of " << i << " is "
                              << std::sqrt(i) << '\n';
}
```

Note that neither `'\n'` nor `std::endl` nor `std::flush` writes the output. You really need the destructor. If we create the synchronized output stream outside the loop, the whole output of any thread is printed together when the destructor is reached.

However, there is a new manipulator for writing the output before the destructor is called: `std::flush_emit`. You can thus create and initialize the synchronized output stream and emit your output line by line also as follows:

```
std::osyncstream coutSync{std::cout};
for (int i = 0; i < num ; ++i) {
  coutSync << "squareroot of " << i << " is "
           << std::sqrt(i) << '\n' << std::flush_emit;
}
```

13.4.3 Using Synchronized Output Streams for Files

You can also use synchronized output stream for files. Consider the following example:

lib/syncfilestream.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  std::osyncstream syncStrm{strm};
  for (int i = 0; i < num ; ++i) {
    syncStrm << "squareroot of " << i << " is "
             << std::sqrt(i) << '\n' << std::flush_emit;

  }
}
```

```
int main()
{
  std::ofstream fs{"tmp.out"};
  std::jthread t1(squareRoots, std::ref(fs), 5);
  std::jthread t2(squareRoots, std::ref(fs), 5);
  std::jthread t3(squareRoots, std::ref(fs), 5);
}
```

This program uses three concurrent threads to write line by line to the same file opened at the beginning of the program.

Note that each thread uses its own synchronized output stream. However, they all have to use the same file stream. Thus, the program would not work if each thread opens the file itself.

13.4.4 Using Synchronized Output Streams as Output Streams

A synchronized output stream *is a* stream. The class `std::osyncstream` is derived from `std::ostream` (to be precise: as usual for stream classes, the class `std::basic_osyncstream<>` is derived from the class `std::basic_ostream<>`). Therefore, you can also implement the program above as follows:

lib/syncfilestream2.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  for (int i = 0; i < num ; ++i) {
    strm << "squareroot of " << i << " is "
         << std::sqrt(i) << '\n' << std::flush_emit;

  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::osyncstream syncStrm1{fs};
  std::jthread t1(squareRoots, std::ref(syncStrm1), 5);
  std::osyncstream syncStrm2{fs};
  std::jthread t2(squareRoots, std::ref(syncStrm2), 5);
  std::osyncstream syncStrm3{fs};
  std::jthread t3(squareRoots, std::ref(syncStrm3), 5);
}
```

The manipulator `std::flush_emit` is defined for output streams in general and can be used here. For output streams that are not synchronized, it has no effect.

Note that creating ***one*** synchronized output stream and passing it to all three threads would not work because then multiple threads would write to one stream:

```
// undefined behavior (concurrent writes to the same stream):
std::osyncstream syncStrm{fs};
std::jthread t1(squareRoots, std::ref(syncStrm), 5);
std::jthread t2(squareRoots, std::ref(syncStrm), 5);
std::jthread t3(squareRoots, std::ref(syncStrm), 5);
```

13.4.5 Synchronized Output Streams in Practice

Since C++20, I use synchronized output stream a lot to "debug" multi-threaded programs with print statements. I only have to define the following:

```
#include <iostream>    // for std::cout
#include <syncstream> // for std::osyncstream

inline auto syncOut(std::ostream& strm = std::cout) {
  return std::osyncstream{strm};
}
```

With this definition, I simply use `syncOut()` instead of `std::cout` to ensure that concurrent output is written line by line. For example:

```
void foo(std::string name) {
  syncOut() << "calling foo(" << name
            << ") in thread " << std::this_thread::get:id() << '\n';
  ...
}
```

We use this in this book to visualize concurrent coroutine output.

To enable me to turn debugging output like this off, I sometimes do the following:

```
#include <iostream>
#include <syncstream> // for std::osyncstream

constexpr bool debug = true;    // switch to false to disable output

inline auto coutDebug() {
  if constexpr (debug) {
    return std::osyncstream{std::cout};
  }
  else {
    struct devnullbuf : public std::streambuf {
      int_type overflow (int_type c) {  // basic output primitive
        return c;                       // - without any print statement
      }
    };
```

```
      static devnullbuf devnull;
      return std::ostream{&devnull};
    }
  }
```

13.5 Afternotes

Latches and barriers were first proposed by Alasdair Mackintosh in http://wg21.link/n3600. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6.

Semaphores were first proposed by Olivier Giroux in http://wg21.link/p0514r1. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6.

Atomic references were first proposed as `std::atomic_view<>` by H. Carter Edwards, Hans Boehm, Olivier Giroux, and James Reus in http://wg21.link/p0019r0. The finally accepted wording was formulated by Daniel Sunderland, H. Carter Edwards, Hans Boehm, Olivier Giroux, Mark Hoemmen, D. Hollman, Bryce Adelstein Lelbach, and Jens Maurer in http://wg21.link/p0019r8. The API for `wait()` and notification was added as finally proposed by David Olsen in http://wg21.link/p1643r1.

Atomic shared pointers were first proposed as `atomic_shared_ptr<>` by Herb Sutter in http://wg21.link/n4058 and were adopted then for the concurrency TS (http://wg21.link/n4577). The finally accepted wording for integrating them into the C++ standard as a partial specialization of `std::atomic<>` was formulated by Alisdair Meredith in http://wg21.link/p0718r2.

Atomic specializations for floating-point types were first proposed by H. Carter Edwards, Hans Boehm, Olivier Giroux, JF Bastien, and James Reus in http://wg21.link/p0020r0. The finally accepted wording was formulated by H. Carter Edwards, Hans Boehm, Olivier Giroux, JF Bastien, and James Reus in http://wg21.link/p0020r6.

Thread synchronization with atomic types was first proposed by Olivier Giroux in http://wg21.link/p0514r0. The finally accepted wording was formulated by Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen in http://wg21.link/p1135r6, http://wg21.link/p1643r1, and http://wg21.link/p1644r0.

Synchronized output streams were first proposed by Lawrence Crowl in http://wg21.link/n3750. The finally accepted wording was formulated by Lawrence Crowl, Peter Sommerlad, Nicolai Josuttis, and Pablo Halpern in http://wg21.link/p0053r7 and by Peter Sommerlad and Pablo Halpern in http://wg21.link/p0753r2.

This page is intentionally left blank

Chapter 14

Coroutines

C++20 introduces support for *coroutines*. Coroutines (invented in 1958 by Mel Conway) are functions you can suspend. This chapter explains the general concepts and most important details of coroutines.

C++20 starts with basic support for coroutines by introducing some core language features and some basic library features for dealing with them. However, significant glue code has to be written to be able to deal with coroutines, which makes using them very flexible but also requires some effort even in simple cases. The plan is to provide more standard types and functions for typical applications of coroutines in the C++ standard library with C++23 and later versions.

This chapter was written with tremendous help and support from Lewis Baker, Frank Birbacher, Michael Eiler, Björn Fahller, Dietmar Kühl, Phil Nash, and Charles Tolman.

14.1 What Are Coroutines?

While ordinary functions (or procedures) are called and then run until their end (or until a return statement is reached or an exception is thrown), coroutines are functions that can run in multiple steps (see Figure 14.1). At certain moments, you can *suspend* a coroutine, which means that the function pauses its computation until it is *resumed*. You might suspend it because the function has to wait for something, there are other (more important) things to do, or you have an intermediate result to yield to the caller.

Starting a coroutine therefore means starting another function until a ***part*** of it is done. The calling function and the coroutine both run switching back and forth between their two paths of execution. Note that both functions do ***not*** run in parallel. Instead, we play a little ping-pong with the control flows:

- A function can decide to ***start*** or ***resume*** its current control flow by starting or continuing with the statements of the coroutine.
- When a coroutine then runs, the coroutine can decide to ***suspend*** or ***end*** its execution, which means that the function that started or resumed the coroutine continues with its control flow.

In the most simplest form of a coroutine, both the main control flow and the control flow of the coroutine run in the same thread. We do not have to use multi-threading and we do not have to deal with concurrent access. However, it is possible to run coroutines in different threads. You can even resume a coroutine on a

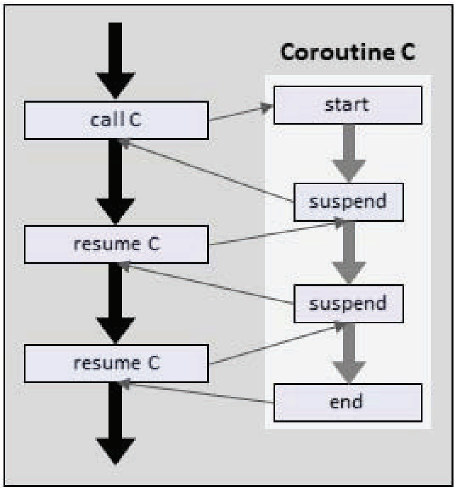


Figure 14.1. Coroutines

different thread to where it was previously suspended. Coroutines are an orthogonal feature which, however, could be used together with multiple threads.

Effectively, using a coroutine is like having a function in the background that you start and continue from time to time. However, because the lifetime of a coroutine goes beyond nested scopes, a coroutine is also an object that stores its state in some memory and provides an API for dealing with it.

There are a few basic aspects in C++ regarding coroutines:

- Coroutines are defined implicitly just by using one of the following keywords in a function:
  - **`co_await`**
  - **`co_yield`**
  - **`co_return`**

  In case none of these keywords is necessary inside a coroutine, you have to explicitly write a `co_return;` statement at the end.
- A coroutine usually returns an object that serves as the ***coroutine interface*** for the caller. Depending on the purpose and use of the coroutine, that object can represent a running task that suspends or switches context from time to time, a generator that yields values from time to time, or a factory that returns one or more values lazily and on demand.
- Coroutines are stackless. You cannot suspend an inner coroutine that was called within an outer coroutine without suspending the outer coroutine. You can only suspend the outer coroutine as a whole.

  When a coroutine is suspended, the state of the coroutine as a whole is stored in an object separately from the stack so that it can be resumed in a totally different context (in a different call stack, in another thread, etc.).

14.2 A First Coroutine Example

Consider we want the following function to be a coroutine:

```
void coro(int max)
{
  std::cout << "CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    std::cout << "CORO " << val << '/' << max << '\n';
  }

  std::cout << "CORO " << max << " end\n";
}
```

This function has a parameter for the maximum value, which we first print. Then, it loops from 1 to this maximum value and prints each value. Finally, we have a print statement at the end.

When the function is called with `coro(3)`, it has the following output:

```
CORO 3 start
CORO 1/3
CORO 2/3
CORO 3/3
CORO 3 end
```

However, we want to program this as a coroutine that is suspended each time the print statement inside the loop is performed. Therefore, the function is interrupted and the user of the coroutine can trigger the next output with a resumption.

14.2.1 Defining the Coroutine

Here is the complete code for the definition of the coroutine:

coro/coro.hpp

```
#include <iostream>
#include "corotask.hpp"    // for CoroTask

CoroTask coro(int max)
{
  std::cout << "         CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "         CORO " << val << '/' << max << '\n';

    co_await std::suspend_always{};     // SUSPEND
  }

  std::cout << "         CORO " << max << " end\n";
}
```

We still have a kind of a function that loops over the values up to the parameter max. However, two things differ from ordinary functions:

- Inside the loop after the print statement, there is a co_await expression, which suspends the coroutine and blocks it until it is resumed. This is called a ***suspend point***.

  The exact behavior of the suspend call is defined by the expression immediately after co_await. It enables programmers to control the exact behavior of the suspension.

  For the moment, we will use a default-constructed object of type std::suspend_always, which accepts the suspension and gives control back to the caller. However, you can reject the suspension or resume another coroutine instead by passing special operands to co_await.
- Although the coroutine has no return statement, it has a return type, CoroTask. This type serves as the *coroutine interface* for the caller of the coroutine. Note that we cannot declare the return type as auto.

  The return type is necessary because the caller needs an interface to be able to deal with the coroutine (such as resuming it). In C++20, coroutine interface types have to be provided by the programmer (or a third party library). We will see later how it is implemented. The plan is that upcoming C++ standards will provide some standard coroutine interface types in their library.

14.2.2 Using the Coroutine

We can use the coroutine as follows:

coro/coro.cpp

```
#include <iostream>
#include "coro.hpp"

int main()
{
  // start coroutine:
  auto coroTask = coro(3);              // initialize coroutine
  std::cout << "coro() started\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.resume()) {           // RESUME
    std::cout << "coro() suspended\n";
  }

  std::cout << "coro() done\n";
}
```

After initializing the coroutine, which yields the coroutine interface coroTask, we start a loop that again and again resumes the coroutine after it has been suspended:

```
auto coroTask = coro(3);              // initialize coroutine

while (coroTask.resume()) {           // RESUME
  ...
}
```

By calling `coro(3)`, we call the coroutine like a function. However, in contrast to function calls, we do not wait for the coroutine to end. Instead, the call returns the coroutine interface to deal with the coroutine after the coroutine is initialized (there is an implicit suspend point at the beginning).

We use `auto` here for the coroutine interface type; however, we could also use its type, which is the return type of the coroutine:

```
CoroTask coroTask = coro(3);              // initialize coroutine
```

The API provided by the class `CoroTask` provides a member function `resume()`, which resumes the coroutine. Each call of `resume()` allows the coroutine to continue to run until the next suspension or until the end of the coroutine. Note that a suspension does not leave any scope in the coroutine. On resumption, we continue the coroutine in the state of its suspension.

The effect is that within the loop in `main()`, we call the next bunch of statements in the coroutine until we reach a suspend point or the end. That is, we call the following:

- First, the initial output, the initialization of `val`, and the first output inside the loop:

  ```
  std::cout << "         CORO " << max << " start\n";

  for (int val = 1; val <= max; ... ) {
      std::cout << "         CORO " << val << '/' << max << '\n';
      ...
  }
  ```

- Then twice, the next iteration in the loop:

  ```
  for (... ; val <= max; ++val) {
      std::cout << "         CORO " << val << '/' << max << '\n';
      ...
  }
  ```

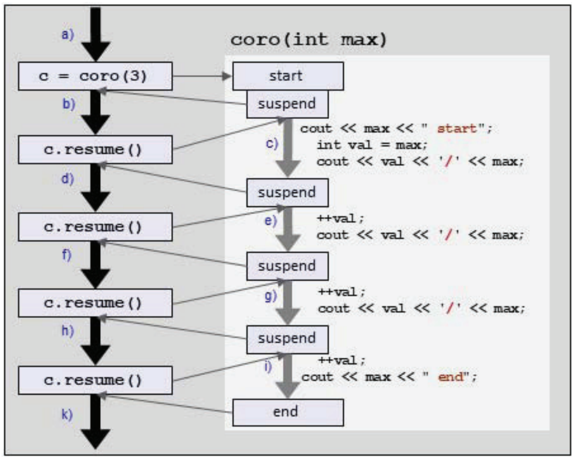
- Finally, after the last iteration in the loop, the coroutine performs the final print statement:

  ```
  for (... ; val <= max; ++val) {
      ...
  }

  std::cout << "         CORO " << max << " end\n";
  ```

The output of the program is as follows:

```
coro() started
         CORO 3 start
         CORO 1/3
coro() suspended
         CORO 2/3
coro() suspended
         CORO 3/3
coro() suspended
         CORO 3 end
coro() done
```



```
cout << max << " start";
int val = max;
cout << val << '/' << max;
```

```
++val;
cout << val << '/' << max;
```

```
++val;
cout << val << '/' << max;
```

```
++val;
cout << max << " end";
```

Figure 14.2. Coroutine example

In combination with the interface type `CoroTask`, which we will see in a moment, we get the following control flow (see Figure 14.2):

a) First, we call the coroutine so that it starts. The coroutine is immediately suspended and the call returns the interface object to deal with the coroutine.

b) We can then use the interface object to resume the coroutine so that it starts performing its statements.

c) Inside the coroutine we process the starting statements up to the first suspend point: the first print statement, the initialization of the local counter `val` in the head of the loop, and (after checking that `val` is less than or equal to `max`) the print statement inside the loop. At the end of this part, the coroutine is suspended.

d) The suspension transfers control back to the main function, which then continues until it resumes the coroutine again.

e) The coroutine continues performing the next statements until we reach the suspend point again, incrementing `val` and (after checking that `val` is still less than or equal to `max`) executing the print statement inside the loop. At the end of this part, the coroutine is suspended again.

   Note that we continue here with the value that `val` had when the coroutine was suspended before.

f) This again transfers control back to the main function, which then continues until it resumes the coroutine once more.

g) The coroutine continues the loop as long as the incremented `val` is less than or equal to `max`.

h) The main function resumes the coroutine as long as the coroutine was suspended with `co_await`.

i) Finally, the coroutine leaves the loop after `val` was incremented, which calls the final print statement with the value of `max`.

j) The end of the coroutine transfers control back to the main function for the last time. The main function then ends its loop and continues until its end.

The exact behavior of the initialization and the interface of the coroutine depends on the interface type `CoroTask`. It might just start the coroutine or provide a context with some initializations, such as opening a file or starting a thread. It also defines whether the coroutine starts immediately or starts lazily (is immediately suspended). With the current implementation of `CoroTask` it starts lazily, which means that the initial call of the coroutine does *not* perform any statements of `coro()` yet.

Note that there is no asynchronous communication or control flow. It is more that `coroTask.resume()` is like a function call for the next portion of `coro()`.

The loop ends when we have reached the end of the coroutine. For this purpose, `resume()` is implemented such that it returns whether the coroutine is not done (yet). Thus, while `resume()` returns `true`, we continue the loop (after printing that it was suspended).

Using Coroutines Multiple Times

As written, a coroutine is kind of a quasi-parallel function, executed sequentially by switching the control flow back and forth. Its state is stored in some heap memory controlled by the coroutine handle, which is usually held by the coroutine interface. By having multiple coroutine interface objects, we can deal with the state of multiple active coroutines, which might be running or suspended independently of each other.

Assume that we start two coroutines with different values for `max`:

coro/coro2.cpp

```cpp
#include <iostream>
#include "coro.hpp"

int main()
{
  // start two coroutines:
  auto coroTask1 = coro(3);             // initialize 1st coroutine
  auto coroTask2 = coro(5);             // initialize 2nd coroutine
  std::cout << "coro(3) and coro(5) started\n";

  coroTask2.resume();                   // RESUME 2nd coroutine once

  // loop to resume the 1st coroutine until it is done:
  while (coroTask1.resume()) {          // RESUME 1st coroutine
    std::cout << "coro() suspended\n";
  }

  std::cout << "coro() done\n";

  coroTask2.resume();                   // RESUME 2nd coroutine again
}
```

We get two different interface objects, `coroTask1` and `coroTask2`, for the initialized coroutines. By resuming sometimes the first and sometimes the second coroutine, our control flow jumps between the main function and these two coroutines.

In our case, we resume the second coroutine (having `5` as `max`) once, then loop over resuming the first coroutine, and finally resume the second coroutine once again. As a result, the program has the following output:

```
coro(3) and coro(5) started
         CORO 5 start
         CORO 1/5
         CORO 3 start
         CORO 1/3
coro() suspended
         CORO 2/3
coro() suspended
         CORO 3/3
coro() suspended
         CORO 3 end
coro() done
         CORO 2/5
```

You can even pass the coroutine interface objects to different functions or threads to resume them there. They will always continue with their current state.

14.2.3 Lifetime Issues with Call-by-Reference

A coroutine usually lives longer than the statement it was initially called from. That has an important consequence: you can run into fatal runtime problems if you pass temporary objects by reference.

Consider the following slightly modified coroutine. All we do now is take the `max` value by reference:

coro/cororef.hpp

```cpp
#include <iostream>
#include <coroutine>        // for std::suspend_always{}
#include "corotask.hpp"     // for CoroTask

CoroTask coro(const int& max)
{
  std::cout << "  CORO " << max << " start\n";   // OOPS: value of max still valid?

  for (int val = 1; val <= max; ++val) {          // OOPS: value of max still valid?
    std::cout << "  CORO " << val << '/' << max << '\n';
    co_await std::suspend_always{};       // SUSPEND
  }

  std::cout << "  CORO " << max << " end\n";    // OOPS: value of max still valid?
}
```

The problem is that passing a temporary object, which might even be a literal, creates undefined behavior. You might or might not see it depending on the platform, compiler settings, and the rest of the code. Consider, for example, calling the coroutine as follows:

coro/cororef.cpp

```
#include <iostream>
#include "cororef.hpp"

int main()
{
  auto coroTask = coro(3);                // OOPS: creates reference to temporary/literal
  std::cout << "coro(3) started\n";
  coro(375);                              // another temporary coroutine
  std::cout << "coro(375) started\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.resume()) {             // ERROR: undefined behavior
    std::cout << "coro() suspended\n";
  }

  std::cout << "coro() done\n";
}
```

On some platforms this code works fine. However, on one platform where I tested this code, I got the following output:

```
coro(3) started
coro(375) started
  CORO -2147168984 start
  CORO -2147168984 end
coro() done
```

After the statement that initializes the coroutine, the location the `max` reference refers to for the passed argument 3 is no longer available. Therefore, when resuming the coroutine for the first time, the output and initialization of `val` are using a reference to a destroyed object.

In general: **do not use references to declare coroutine parameters**.

If copying the parameter becomes too expensive, you can “pass by reference” by using reference wrappers created with `std::ref()` or `std::cref()`. For containers, you can use `std::views::all()` instead, which passes the container as a view, so that all standard range functions can still be used without converting the parameter back. For example:

```
CoroTask printElems(auto coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
    co_await std::suspend_always{};       // SUSPEND
  }
}

std::vector<std::string> coll;
...
// start coroutine that prints the elements:
// - use view created with std::views::all() to avoid copying the container
auto coPrintElems = printElems(std::views::all(coll));

while (coPrintElems.resume()) {             // RESUME
  ...
}
```

14.2.4 Coroutines Calling Coroutines

Coroutines can call other coroutines (even indirectly) and both the calling and the called coroutines might have suspend points. Let us look at what that means.

Consider a coroutine with one suspend point so that it runs in two parts:

```
CoroTask coro()
{
  std::cout << "    coro(): PART1\n";
  co_await std::suspend_always{};      // SUSPEND
  std::cout << "    coro(): PART2\n";
}
```

When we use this coroutine as follows:

```
auto coroTask = coro();                // initialize coroutine
std::cout << "MAIN: coro() initialized\n";

while (coroTask.resume()) {            // RESUME
  std::cout << "MAIN: coro() suspended\n";
}

std::cout << "MAIN: coro() done\n";
```

We get the following output:

```
MAIN: coro() initialized
    coro(): PART1
MAIN: coro() suspended
    coro(): PART2
MAIN: coro() done
```

Now let us call `coro()` indirectly via another coroutine. For that, `main()` calls `callCoro()` instead of `coro()`:

```
auto coroTask = callCoro();             // initialize coroutine
std::cout << "MAIN: callCoro() initialized\n";

while (coroTask.resume()) {             // RESUME
  std::cout << "MAIN: callCoro() suspended\n";
}

std::cout << "MAIN: callCoro() done\n";
```

The interesting part is how to implement `callCoro()`.

No Inner `resume()`

We might try to implement `callCoro()` by just calling `coro()`:

```
CoroTask callCoro()
{
  std::cout << "  callCoro(): CALL coro()\n";
  coro();                               // CALL sub-coroutine
  std::cout << "  callCoro(): coro() done\n";
  co_await std::suspend_always{};       // SUSPEND
  std::cout << "  callCoro(): END\n";
}
```

This compiles. However, it does not work as expected, as the output of the program demonstrates:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

The print statements in the body of `coro()` are never called.

The reason is that

```
coro();
```

only ***initializes*** the coroutine and immediately suspends it. `coro()` is never resumed. It cannot be resumed because the coroutine interface returned is not even used. A `resume()` of the outer coroutine does ***not*** automatically resume any inner coroutine.

To get at least a compiler warning when using coroutines like functions, class `CoroTask` will be declared with `[[nodiscard]]`.

Using an Inner `resume()`

We have to deal with the inner coroutine the same way we deal with the outer coroutine: `resume()` it in a loop:

```
CoroTask callCoro()
{
  std::cout << "  callCoro(): CALL coro()\n";
  auto sub = coro();                      // init sub-coroutine
  while (sub.resume()) {                  // RESUME sub-coroutine
    std::cout << "  callCoro(): coro() suspended\n";
  }
  std::cout << "  callCoro(): coro() done\n";
  co_await std::suspend_always{};         // SUSPEND
  std::cout << "  callCoro(): END\n";
}
```

With this implementation of callCoro(), we get the behavior and output we want:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
  callCoro(): coro() suspended
    coro(): PART2
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

See *coro/corocoro.cpp* for the complete example.

Using One Inner resume()

It is worth noting what happens if we resume coro() inside callCoro() only once:

```
  CoroTask callCoro()
  {
    std::cout << "  callCoro(): CALL coro()\n";
    auto sub = coro();                      // init sub-coroutine
    sub.resume();                           // RESUME sub-coroutine
    std::cout << "  callCoro(): call.resume() done\n";
    co_await std::suspend_always{};         // SUSPEND
    std::cout << "  callCoro(): END\n";
  }
```

The output becomes:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
  callCoro(): call.resume() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

After callCoro() initializes coro(), coro() is resumed only once, which means that only the first part of it is called. After that, the suspension of coro() transfers the control flow back to callCoro(), which then suspends itself, meaning that the control flow goes back to main(). When main() resumes callCoro(), the program finishes callCoro(). coro() is never finished at all.

Delegating resume()

It is possible to implement CoroTask so that co_await registers a sub-coroutine in a way that a suspension in the sub-coroutine is handled like a suspension in the calling coroutine. callCoro() would then look as follows:

```
CoroTaskSub callCoro()
{
  std::cout << "  callCoro(): CALL coro()\n";
  co_await coro();                        // call sub-coroutine
  std::cout << "  callCoro(): coro() done\n";
  co_await std::suspend_always{};         // SUSPEND
  std::cout << "  callCoro(): END\n";
}
```

However, resume() then has to delegate a request for resumption to the sub-coroutine if there is one. For this purpose, the CoroTask interface has to become an *awaitable*. Later, after we have introduced awaitables, we will see such a coroutine interface that delegates resumptions to sub-coroutines.

14.2.5 Implementing the Coroutine Interface

I have stated a couple of times that in our example, class CoroTask plays an important role for dealing with the coroutine. It is ***the*** interface that both the compiler and the caller of a coroutine usually deal with. The coroutine interface brings together a couple of requirements to let the compiler deal with coroutines and provides the API for the caller to create, resume, and destroy the coroutine. Let us elaborate on that in detail.

To deal with coroutines in C++ we need two things:

- A ***promise type***

  This type is used to define certain customization points for dealing with a coroutine. Specific member functions define callbacks that are called in certain situations.
- An internal ***coroutine handle*** of the type std::coroutine_handle<>

  This object is created when a coroutine is called (using one of the standard callbacks of the promise type above). It can be used to manage the state of the coroutine by providing a low-level interface to resume a coroutine as well as to deal with the end of the coroutine.

The usual purpose of the type dealing with the return type of a coroutine is to bring these requirements together:

- It has to define which promise type is used (it is usually defined as the type member promise_type).
- It has to define where the coroutine handle is stored (it is usually defined as a data member).
- It has to provide the interface for the caller to deal with the coroutine (the member function resume() in this case).

The Coroutine Interface `CoroTask`

The class `CoroTask`, which provides `promise_type`, stores the coroutine handle, and defines the API for the caller of the coroutine, may be defined as follows:

coro/corotask.hpp

```
#include <coroutine>

// coroutine interface to deal with a simple task
// - providing resume() to resume the coroutine
class [[nodiscard]] CoroTask {
 public:
   // initialize members for state and customization:
   struct promise_type;      // definition later in corotaskpromise.hpp
   using CoroHdl = std::coroutine_handle<promise_type>;
 private:
   CoroHdl hdl;              // native coroutine handle

 public:
   // constructor and destructor:
   CoroTask(auto h)
    : hdl{h} {               // store coroutine handle in interface
   }
   ~CoroTask() {
     if (hdl) {
       hdl.destroy();        // destroy coroutine handle
     }
   }
   // don't copy or move:
   CoroTask(const CoroTask&) = delete;
   CoroTask& operator=(const CoroTask&) = delete;

   // API to resume the coroutine
   // - returns whether there is still something to process
   bool resume() const {
     if (!hdl || hdl.done()) {
       return false;         // nothing (more) to process
     }
     hdl.resume();           // RESUME (blocks until suspended again or the end)
     return !hdl.done();
   }
};

#include "corotaskpromise.hpp"    // definition of promise_type
```

In the class `CoroTask`, we first define the basic types and member to deal with the native API of the coroutine. The key member the compiler looks for in a coroutine interface type is the type member `promise_type`. With the promise type, we define the type of the coroutine handle and introduce a private member where this coroutine handle, specialized for the promise type, is stored:

```
class [[nodiscard]] CoroTask {
 public:
  // initialize members for state and customization:
  struct promise_type;      // definition later in corotaskpromise.hpp
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;              // native coroutine handle
  ...
};
```

We introduce the `promise_type` (which every coroutine type has to have) and declare the native coroutine handle `hdl`, which manages the state of the coroutine. As you can see, the type of the native coroutine handle, `std::coroutine_handle<>`, is parameterized with the promise type. That way, any data stored in the promise is part of the handle and the functions in the promise are available via the handle.

The promise type has to be public to be visible from the outside. It is often also helpful to provide a public name of the type of the coroutine handle (`CoroHdl` in this case). To simplify things, we could even make the handle itself public. That way, we can use

```
CoroTask::CoroHdl
```

instead of

```
std::coroutine_handle<CoroTask::promise_type>
```

We could also define the promise type here directly inline. However, in this example, we defer the definition to the later included `corotaskpromise.hpp`.

The constructor and destructor of the coroutine interface type initialize the member for the coroutine handle and clean it up before the coroutine interface is destroyed:

```
class CoroTask {
  ...
 public:
  CoroTask(auto h)
   : hdl{h} {               // store coroutine handle internally
  }
  ~CoroTask() {
    if (hdl) {
      hdl.destroy();        // destroy coroutine handle (if there is one)
    }
  }
  ...
};
```

By declaring the class with `[[nodiscard]]`, we force compiler warnings when the coroutine is created but not used (which might especially happen by accidentally using a coroutine as an ordinary function).

For simplicity, we disable copying and moving. Providing copy or move semantics is possible but you have to be careful to deal with the resources correctly.

Finally, we define our only interface for the caller, `resume()`:

```
class CoroTask {
  ...
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;        // nothing (more) to process
    }
    hdl.resume();          // RESUME (blocks until suspended again or the end)
    return !hdl.done();
  }
};
```

The key API is `resume()`, which resumes the coroutine when it is suspended. It more or less propagates the request to resume to the native coroutine handle. It also returns whether it makes sense to resume the coroutine again.

First, the function checks whether we have a handle at all or whether the coroutine has already ended. Even though in this implementation the coroutine interface always has a handle, this is a safety net which is necessary, for example, if the interface supports move semantics.

Calling `resume()` is only allowed when the coroutine is suspended and has not reached its end. Therefore, checking whether it is `done()` is necessary. The call itself resumes the suspended coroutine and blocks until it reaches the next suspend point or the end.

```
    hdl.resume();          // RESUME (blocks until suspended again or the end)
```

You can also use `operator()` for this:

```
    hdl();                 // RESUME (blocks until suspended again or the end)
```

Because our `resume()` interface for the caller returns whether it makes sense to resume the coroutine again, we return whether the coroutine has reached its end:

```
  bool resume() const {
    ...
    return !hdl.done();
  }
```

The member function `done()` is provided by the native coroutine handle just for that purpose.

As you can see, the interface for the caller of the coroutine wraps the native coroutine handle and its API completely. In this API, we decide how the caller can deal with the coroutine. We could use different function names or even operators or split the calls for resumption and checking for the end. Later on you will see examples where we iterate over values the coroutine delivers with each suspension, we can even send values back to the coroutine, and we can provide an API to put the coroutine in a different context.

Finally, we include a header file with the definition of the promise type:

```
#include "corotaskpromise.hpp"    // definition of promise_type
```

This is usually done inside the declaration of the interface class or at least in the same header file. However, this way, we can split the details of this example over different files.

Implementing `promise_type`

The only missing piece is the definition of the promise type. Its purpose is to:

- Define how to create or get the return value of the coroutine (which usually includes creating the coroutine handle)
- Decide whether the coroutine should suspend at its beginning or end
- Deal with values exchanged between the caller of the coroutine and the coroutine
- Deal with unhandled exceptions

Here is a typical basic implementation of the promise type for `CoroTask` and its coroutine handle type `CoroHdl`:

coro/corotaskpromise.hpp

```
struct CoroTask::promise_type {
  auto get_return_object() {         // init and return the coroutine interface
    return CoroTask{CoroHdl::from_promise(*this)};
  }
  auto initial_suspend() {           // initial suspend point
    return std::suspend_always{};    // - suspend immediately
  }
  void unhandled_exception() {       // deal with exceptions
    std::terminate();                // - terminate the program
  }
  void return_void() {               // deal with the end or co_return;
  }
  auto final_suspend() noexcept {    // final suspend point
    return std::suspend_always{};    // - suspend immediately
  }
};
```

We (have to) define the following members (using the typical order in which they are used):

- **`get_return_object()`** is called to initialize the coroutine interface. It creates the object that is later returned to the caller of the coroutine. It is usually implemented as follows:
  - First, it creates the native coroutine handle for the promise that we call this function for:

    coroHdl = `CoroHdl::from_promise(*this)`

    The promise that we call this member function for is automatically created when we start the coroutine.
    `from_promise()` is a static member function that the class template `std::coroutine_handle<>` provides for this purpose.

  - Then, we create the coroutine interface object, initializing it with the handle just created:

    coroIf `= CoroTask{`*coroHdl*`}`

  - Finally, we return the interface object:

    `return` *coroIf*

  The implementation does this all in one statement:

```
auto get_return_object() {
  return CoroTask{CoroHdl::from_promise(*this)};
}
```

  We could also return the coroutine handle here without explicitly creating the coroutine interface from it:

```
auto get_return_object() {
  return CoroHdl::from_promise(*this);
}
```

  Internally, the returned coroutine handle is then automatically converted by using it to initialize the coroutine interface. However, this approach is not recommended, because it does not work if the constructor of `CoroTask` is `explicit` and because it is not clear when the interface is created.

- **`initial_suspend()`** allows additional initial preparations and defines whether the coroutine should start eagerly or lazily:
  - Returning `std::suspend_never{}` means starting eagerly. The coroutine starts immediately after its initialization with the first statements.
  - Returning `std::suspend_always{}` means starting lazily. The coroutine is suspended immediately without executing any statement. It is processed with the resumption.

  In our example, we request immediate suspension.

- **`return_void()`** defines the reaction when reaching the end (or a `co_return;` statement). it this member function is declared, the coroutine should never return a value. If the coroutine yields or returns data, you have to use another member function instead.

- **`unhandled_exception()`** defines how to deal with exceptions not handled locally in the coroutine. Here, we specify that this results in an abnormal termination of the program. Other ways of dealing with exceptions are discussed later.

- **`final_suspend()`** defines whether the coroutine should be finally suspended. Here, we specify that we want to do so, which is usually the right thing to do. Note that this member function has to guarantee not to throw exceptions. It should always return `std::suspend_always{}`.

The purpose and use of these members of the promise type are discussed in detail later.

14.2.6 Bootstrapping Interface, Handle, and Promise

Let us recapitulate what we need to deal with a coroutine:

- For each coroutine we have a ***promise***, which is created automatically when a coroutine is called.
- The coroutine state is stored in a ***coroutine handle***. It has the type `std::coroutine_handle<`*PrmType*`>`. That type provides the native API to resume a coroutine (and check whether it is at its end or destroy its memory).
- The ***coroutine interface*** is the typical place to bring everything together. It holds and manages the native coroutine handle, it is returned by the call of the coroutine, and it provides member functions to deal with the coroutine.

There are various ways to declare the promise type and the coroutine handle (and the type of both). There is no obvious way to do it because the type of the coroutine handle needs the promise type and the definition of the promise type uses the coroutine handle.

In practice, the following options are usually used:

- Declare the promise type, declare the type of the coroutine handle, and define the promise type:

```
class CoroTask {
 public:
  struct promise_type;                        // promise type
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;                                // native coroutine handle
 public:
  struct promise_type {
    auto get_return_object() {
      return CoroTask{CoroHdl::from_promise(*this)};
    }
    ...
  };
  ...
};
```

- Define the promise type and declare the coroutine handle:

```
class CoroTask {
 public:
  struct promise_type {                       // promise type
    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    ...
  };
 private:
  std::coroutine_handle<promise_type> hdl;    // native coroutine handle
 public:
  ...
};
```

- Define the promise type outside as a generic helper type:

```
template<typename CoroIf>
struct CoroPromise {
  auto get_return_object() {
    return std::coroutine_handle<CoroPromise<CoroIf>>::from_promise(*this);
  }
  ...
};

class CoroTask {
 public:
  using promise_type = CoroPromise<CoroTask>;
 private:
  std::coroutine_handle<promise_type> hdl;       // native coroutine handle
 public:
  ...
};
```

Because the promise type is often interface-specific (having different/additional members), I usually use the following very condensed form:

```
class CoroTask {
 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;                                  // native coroutine handle
 public:
  struct promise_type {
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
    ...
  };
  ...
};
```

Note that everything described so far is kind of the ***typical*** way to deal with coroutines. There is more flexibility designed into the coroutine library. For example:

- You can store or manage coroutine interfaces in a container or scheduler.
- You can even skip the coroutine interface completely. However, this is an option that is very rarely used.

14.2.7 Memory Management

Coroutines have a state that is used in different contexts. For this purpose, a coroutine handle usually stores the state of a coroutine in heap memory. The heap memory allocation can be optimized away or pragmatically changed.

Calling `destroy()`

To make coroutine handles cheap, there is no smart handling of this memory. Coroutine handles just point to the memory when initialized until `destroy()` is called. Therefore, you should usually call `destroy()` explicitly when the coroutine interface is destroyed:

```
class CoroTask {
  ...
  CoroHdl hdl;                  // native coroutine handle

 public:
  ~CoroTask() {
    if (hdl) {
      hdl.destroy();            // destroy coroutine handle (if there is one)
    }
  }
  ...
};
```

Copying and Moving Coroutines

The cheap/naive implementation of coroutine handles also makes it necessary to deal with copying and moving. By default, copying the coroutine interface would copy the coroutine handle, which would have the effect that two coroutine interfaces/handles share the same coroutine. This, of course, introduces risks when one coroutine handle brings the coroutine into a state that the other handle is not aware of. Moving coroutine objects has the same effect, because by default, pointers are copied with a move.

To reduce the danger, you should be careful about providing copy or move semantics for coroutines. The easiest approach is to disable copying and moving:

```
class CoroTask {
  ...
  // don't copy or move:
  CoroTask(const CoroTask&) = delete;
  CoroTask& operator=(const CoroTask&) = delete;
  ...
};
```

However, this then means that you cannot move coroutines around (such as storing them in a container). Therefore, it might make sense to support move semantics. However, you should ensure that the moved-from object no longer refers to the coroutine and that a coroutine that gets a new value destroys the data for the old one:

```
class CoroTask {
  ...
  // support move semantics:
  CoroTask(CoroTask&& c) noexcept
   : hdl{std::move(c.hdl)} {
    c.hdl = nullptr;
  }
  CoroTask& operator=(CoroTask&& c) noexcept {
    if (this != &c) {           // if no self-assignment
      if (hdl) {
        hdl.destroy();          // - destroy old handle (if there is one)
      }
      hdl = std::move(c.hdl);   // - move handle
      c.hdl = nullptr;          // - moved-from object has no handle anymore
    }
    return *this;
  }
  ...
};
```

Strictly speaking, you do not need `std::move()` for the handle here, but it does not hurt and reminds you that we delegate the move semantics to the member.

14.3 Coroutines That Yield or Return Values

After introducing a first example using `co_await`, we should also introduce the other two keywords for coroutines:

- `co_yield` allows coroutines to yield a value each time they are suspended.
- `co_return` allows coroutines to return a value at their end.

14.3.1 Using `co_yield`

By using `co_yield`, a coroutine can yield intermediate results when it is suspended.

An obvious example is a coroutine that "generates" values. As a variation of our example, we could loop over some values up to `max` and yield them to the caller of the coroutine instead of only printing them. For this purpose, the coroutine looks as follows:

coro/coyield.hpp

```
#include <iostream>
#include "corogen.hpp"     // for CoroGen

CoroGen coro(int max)
{
  std::cout << "         CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "         CORO " << val << '/' << max << '\n';

    // yield next value:
    co_yield val;                      // SUSPEND with value
  }

  std::cout << "         CORO " << max << " end\n";
}
```

By using `co_yield`, the coroutine yields intermediate results. When the coroutine reaches `co_yield`, it suspends the coroutine, providing the value of the expression behind `co_yield`:

```
co_yield val;    // calls yield_value(val) on promise
```

The coroutine frame maps this to a call of `yield_value()` for the promise of the coroutine, which can define how to handle this intermediate result. In our case, we store the value in a member of the promise, which makes it available in the coroutine interface:

```
struct promise_type {
  int coroValue = 0;               // last value from co_yield

  auto yield_value(int val) {      // reaction to co_yield
```

```
      coroValue = val;              // - store value locally
      return std::suspend_always{};  // - suspend coroutine
    }
    ...
  };
```

After storing the value in the promise, we return `std::suspend_always{}`, which really suspends the coroutine. We could program different behavior here, so that the coroutine (conditionally) continues.

The coroutine can be used as follows:

coro/coyield.cpp

```cpp
#include "coyield.hpp"
#include <iostream>

int main()
{
  // start coroutine:
  auto coroGen = coro(3);                // initialize coroutine
  std::cout << "coro() started\n";

  // loop to resume the coroutine until it is done:
  while (coroGen.resume()) {             // RESUME
    auto val = coroGen.getValue();
    std::cout << "coro() suspended with " << val << '\n';
  }

  std::cout << "coro() done\n";
}
```

Again, by calling `coro(3)` we initialize the coroutine to count until 3. Whenever we call `resume()` for the returned coroutine interface, the coroutine "computes" and yields the next value.

However, `resume()` does not return the yielded value (it still returns whether it makes sense to resume the coroutine again). To access the next value, `getValue()` is provided and used. Therefore, the program has the following output:

```
coro() started
         CORO 3 start
         CORO 1/3
coro() suspended with 1
         CORO 2/3
coro() suspended with 2
         CORO 3/3
coro() suspended with 3
         CORO 3 end
coro() done
```

The coroutine interface has to deal with the yielded values and provides a slightly different API for the caller. Therefore, we use a different type name: CoroGen. The name demonstrates that instead of performing a *task*, which we suspend from time to time, we have a coroutine that *generates* values with each suspension. The type CoroGen might be defined as follows:

coro/corogen.hpp

```cpp
#include <coroutine>
#include <exception>  // for terminate()

class [[nodiscard]] CoroGen {
 public:
  // initialize members for state and customization:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;                           // native coroutine handle
 public:
  struct promise_type {
    int coroValue = 0;                   // recent value from co_yield

    auto yield_value(int val) {          // reaction to co_yield
      coroValue = val;                   // - store value locally
      return std::suspend_always{};      // - suspend coroutine
    }

    // the usual members:
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

  // constructors and destructor:
  CoroGen(auto h) : hdl{h} { }
  ~CoroGen() { if (hdl) hdl.destroy(); }

  // no copying or moving:
  CoroGen(const CoroGen&) = delete;
  CoroGen& operator=(const CoroGen&) = delete;

  // API:
  // - resume the coroutine:
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;      // nothing (more) to process
    }
    hdl.resume();        // RESUME
    return !hdl.done();
  }

  // - yield value from co_yield:
  int getValue() const {
    return hdl.promise().coroValue;
  }
};
```

In general, the definition of the coroutine interface used here follows the general principles of coroutine interfaces. Two things differ:

- The promise provides the member `yield_value()`, which is called whenever `co_yield` is reached.
- For the external API of the coroutine interface, `CoroGen` provides `getValue()`. It returns the last yielded value stored in the promise:

```
class CoroGen {
 public:
  ...
  int getValue() const {
    return hdl.promise().coroValue;
  }
};
```

Iterating Over Values That a Coroutine Yields

We could also use a coroutine interface that can be used like a range (providing an API to iterate over the values that suspensions yield):

coro/cororange.cpp

```
#include "cororange.hpp"
#include <iostream>
#include <vector>

int main()
{
  auto gen = coro(3);              // initialize coroutine
  std::cout << "--- coro() started\n";

  // loop to resume the coroutine for the next value:
  for (const auto& val : gen) {
    std::cout << "    val: " << val << '\n';
  }
  std::cout << "--- coro() done\n";
}
```

We only need a slightly different generator returned by the coroutine (see *coro/cororange.hpp*):

```
Generator<int> coro(int max)
{
  std::cout << "CORO " << max << " start\n";
  ...
}
```

As you can see, we use a generic coroutine interface for generator values of a passed type (`int` in this case). A very simple implementation (which shows the point but has some flaws) might look as follows:

coro/generator.hpp

```
#include <coroutine>
#include <exception>   // for terminate()
#include <cassert>     // for assert()

template<typename T>
class [[nodiscard]] Generator {
 public:
  // customization points:
  struct promise_type {
    T coroValue{};                          // last value from co_yield

    auto yield_value(T val) {                // reaction to co_yield
      coroValue = val;                       // - store value locally
      return std::suspend_always{};          // - suspend coroutine
    }

    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

 private:
  std::coroutine_handle<promise_type> hdl;    // native coroutine handle

 public:
  // constructors and destructor:
  Generator(auto h) : hdl{h} { }
  ~Generator() { if (hdl) hdl.destroy(); }

  // no copy or move supported:
  Generator(const Generator&) = delete;
  Generator& operator=(const Generator&) = delete;

  // API to resume the coroutine and access its values:
  // - iterator interface with begin() and end()
  struct iterator {
    std::coroutine_handle<promise_type> hdl;  // nullptr on end
    iterator(auto p) : hdl{p} {
    }
    void getNext() {
      if (hdl) {
        hdl.resume();         // RESUME
        if (hdl.done()) {
          hdl = nullptr;
        }
      }
    }
    int operator*() const {
      assert(hdl != nullptr);
      return hdl.promise().coroValue;
    }
    iterator operator++() {
      getNext();              // resume for next value
      return *this;
    }
    bool operator== (const iterator& i) const = default;
  };

  iterator begin() const {
    if (!hdl || hdl.done()) {
      return iterator{nullptr};
    }
    iterator itor{hdl};       // initialize iterator
    itor.getNext();           // resume for first value
    return itor;
  }

  iterator end() const {
    return iterator{nullptr};
  }
};
```

The key approach is that the coroutine interface provides the members `begin()` and `end()` and an iterator type `iterator` to iterate over the values:

- `begin()` yields the iterator after resuming for the first value. The iterator stores the coroutine handle internally so that it knows its state.
- `operator++()` of the iterator then yields the next value.
- `end()` yields a state that represents the end of the range. Its `hdl` is `nullptr`. This is also the state an iterator gets when there is no more value.
- `operator==()` of the iterator is default generated just comparing the state by comparing the handles of two iterators.

C++23 will probably provide such a coroutine interface in its standard library as `std::generator<>` with a far more sophisticated and robust API (see `http://wg21.link/p2502`).

14.3.2 Using `co_return`

By using `co_return`, a coroutine can return a result to the caller at its end.

Consider the following example:

coro/coreturn.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
#include <coroutine>         // for std::suspend_always{}
#include "resulttask.hpp"    // for ResultTask

ResultTask<double> average(auto coll)
{
  double sum = 0;
  for (const auto& elem : coll) {
    std::cout << "  process " << elem << '\n';
    sum = sum + elem;
    co_await std::suspend_always{};          // SUSPEND
  }
  co_return sum / std::ranges::ssize(coll);  // return resulting average
}

int main()
{
  std::vector values{0, 8, 15, 47, 11, 42};

  // start coroutine:
  auto task = average(std::views::all(values));

  // loop to resume the coroutine until all values have been processed:
  std::cout << "resume()\n";
  while (task.resume()) {                    // RESUME
    std::cout << "resume() again\n";
  }

  // print return value of coroutine:
  std::cout << "result: " << task.getResult() << '\n';
}
```

In this program, `main()` starts a coroutine `average()`, which iterates over the elements of the passed collection and adds their values to an initial `sum`. After processing each element, the coroutine is suspended. At the end, the coroutine returns the average by dividing the `sum` by the number of elements.

Note that you need `co_return` to return the result of the coroutine. Using `return` is not allowed.

The coroutine interface is defined in the class `ResultTask`, which is parameterized for the type of the return value. This interface provides `resume()` to resume the coroutine whenever it is suspended. In addition, it provides `getResult()` to ask for the return value of the coroutine after it is done.

The coroutine interface `ResultTask<>` looks as follows:

coro/resulttask.hpp

```
#include <coroutine>
#include <exception>   // for terminate()

template<typename T>
class [[nodiscard]] ResultTask {
 public:
  // customization points:
  struct promise_type {
    T result{};                              // value from co_return

    void return_value(const auto& value) { // reaction to co_return
      result = value;                        // - store value locally
    }

    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    auto initial_suspend() { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };
 private:
  std::coroutine_handle<promise_type> hdl;   // native coroutine handle

 public:
  // constructors and destructor:
  // - no copying or moving is supported
  ResultTask(auto h) : hdl{h} { }
  ~ResultTask() { if (hdl) hdl.destroy(); }
  ResultTask(const ResultTask&) = delete;
  ResultTask& operator=(const ResultTask&) = delete;

  // API:
  // - resume() to resume the coroutine
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;      // nothing (more) to process
    }
    hdl.resume();        // RESUME
    return !hdl.done();
  }

  // - getResult() to get the last value from co_yield
  T getResult() const {
    return hdl.promise().result;
  }
};
```

Again, the definition follows the general principles of coroutine interfaces.

However, this time we have support for a return value. Therefore, in the promise type, the customization point `return_void()` is no longer provided. Instead, `return_value()` is provided, which is called when the coroutine reaches the `co_return` expression:

```
template<typename T>
class ResultTask {
 public:
  struct promise_type {
    T result{};                                  // value from co_return

    void return_value(const auto& value) {       // reaction to co_return
      result = value;                            // - store value locally
    }
    ...
  };
  ...
};
```

The coroutine interface then just returns this value whenever `getResult()` is called:

```
template<typename T>
class ResultTask {
 public:
  ...
```

```
    T getResult() const {
      return hdl.promise().result;
    }
  };
```

return_void() and return_value()

Note that it is undefined behavior if coroutines are implemented in a way that they sometimes may and sometimes may not return a value. This coroutine is not valid:

```
  ResultTask<int> coroUB(...)
  {
    if (...) {
      co_return 42;
    }
  }
```

14.4 Coroutine Awaitables and Awaiters

So far, we have seen how coroutines are controlled from the outside using coroutine interfaces (wrapping coroutine handles and their promises). However, there is another configuration point that coroutines can (and have to) provide themselves: ***awaitables*** (with ***awaiters*** as special way to implement them).

The terms are correlated as follows:

- ***Awaitables*** is the term for what the operator `co_await` needs as its operand. Thus, *awaitables* are all objects that `co_await` can deal with.
- ***Awaiter*** is the term for one specific (and typical) way to implement an awaitable. It has to provide three specific member functions to deal with the suspension and the resumption of a coroutine.

Awaitables are used whenever `co_await` (or `co_yield`) is called. They allow the provision of code that rejects the request to suspend (temporarily or on certain conditions) or execute some logic for suspensions and resumptions.

You have used two awaiter types already: `std::suspend_always` and `std::suspend_never`.

14.4.1 Awaiters

Awaiters can be called when a coroutine is suspended or resumed. Table *Special member functions of awaiters* lists the required operations that awaiters have to provide.

Operation	**Effect**
`await_ready()`	Yields whether suspension is (currently) disabled
`await_suspend(`*awaitHdl*`)`	Handle suspension
`await_resume()`	Handle resumption

Table 14.1. Special member functions of awaiters

The key functions are `await_suspend()` and `await_resume()`:

- **`await_ready()`**

  This function is called for a coroutine immediately before the coroutine is suspended. It is provided to (temporarily) turn off suspension completely. If it returns `true`, the coroutine is *not* suspended at all.

  This function usually just returns `false` ("no, do not block/ignore any suspension"). To save the cost of suspension, it might conditionally yield `true` when a suspension makes no sense (e.g., if the suspension depends on some data being available).

  Note that in this function, the coroutine is ***not*** suspended yet. It should not be used to directly or indirectly call `resume()` or `destroy()` for the coroutine it is called for.

- **`auto await_suspend(`*awaitHdl*`)`**

  This function is called for a coroutine immediately after the coroutine is suspended.

  The parameter *awaitHdl* is the handle of the coroutine that was suspended. It has the type `std::coroutine_handle<`*PromiseType*`>`.

  In this function, you can specify what to do next, including resuming the suspended or the awaiting coroutine immediately. Different return types allow you to specify this in different ways. You can also effectively skip the suspension by directly transferring control to another coroutine.

You could even destroy the coroutine here. In this case, however, make sure that you do not use the coroutine anywhere else anymore (such as calling `done()` or `destroy()` in the coroutine interface).

- **`auto await_resume()`**
  This function is called for a coroutine when the coroutine is resumed after a successful suspension.
  It can return a value, which would be the value the `co_await` expression yields.

Consider the following trivial awaiter:

coro/awaiter.hpp

```
#include <iostream>

class Awaiter {
 public:
  bool await_ready() const noexcept {
    std::cout << "   await_ready()\n";
    return false;         // do suspend
  }

  void await_suspend(auto hdl) const noexcept {
    std::cout << "   await_suspend()\n";
  }

  void await_resume() const noexcept {
    std::cout << "   await_resume()\n";
  }
};
```

In this awaiter, we trace when which function of the awaiter is called. Because `await_ready()` returns `false` and `await_suspend()` returns nothing, the awaiter accepts the suspension (without resuming something else). This is the behavior of the standard awaiter `std::suspend_always{}` combined with some print statements. Other return types/values can provide different behavior.

A coroutine might use this awaiter as follows (see *coro/awaiter.cpp* for the complete code):

```
CoroTask coro(int max)
{
  std::cout << "  CORO start\n";
  for (int val = 1; val <= max; ++val) {
    std::cout << "  CORO " << val << '\n';
    co_await Awaiter{};  // SUSPEND with our own awaiter
  }
  std::cout << "  CORO end\n";
}
```

Assume we use the coroutine as follows:

```
auto coTask = coro(2);
std::cout << "started\n";

std::cout << "loop\n";
while (coTask.resume()) {  // RESUME
  std::cout << " suspended\n";
}
std::cout << "done\n";
```

As a result, we get the following output:

```
started
loop
  CORO start
  CORO 1
   await_ready()
   await_suspend()
 suspended
   await_resume()
  CORO 2
   await_ready()
   await_suspend()
 suspended
   await_resume()
  CORO end
done
```

We will discuss more details of awaiters later.

14.4.2 Standard Awaiters

The C++ standard library provides two trivial awaiters we have already used:

- `std::suspend_always`
- `std::suspend_never`

Their definitions are pretty straightforward:

```
namespace std {
  struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept { }
    constexpr void await_resume() const noexcept { }
  };

  struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept { }
```

```
      constexpr void await_resume() const noexcept { }
    };
  }
```

Both do nothing on suspension or resumption. However, their return values of `await_ready()` differ:

- If we return `false` in `await_ready()` (and nothing in `await_suspend()`), `suspend_always` accepts every suspension, which means that the coroutine returns back to its caller.
- If we return `true` in `await_ready()`, `suspend_never` never accepts any suspension, which means that the coroutine continues (`await_suspend()` is never called).

As seen, the awaiters are typically used as basic awaiters for `co_await`:

```
  co_await std::suspend_always{};
```

They are also used by returning them in `initial_suspend()` and `finally_suspend()` of coroutine promises.

14.4.3 Resuming Sub-Coroutines

By making a coroutine interface an awaitable (providing the awaiter API), we allow it to handle sub-coroutines in a way that their suspend points become suspend points of the main coroutine.

This allows programmers to avoid having nested loops for resumption, as introduced by the example *corocoro.cpp*.

Taking the first implementation of `CoroTask`, we only have to provide the following modifications:

- The coroutine interface has to know its sub-coroutine (if there is one):

```
    class CoroTaskSub {
     public:
      struct promise_type;
      using CoroHdl = std::coroutine_handle<promise_type>;
     private:
      CoroHdl hdl;                 // native coroutine handle
     public:
      struct promise_type {
        CoroHdl subHdl = nullptr;          // sub-coroutine (if there is one)
        ...
      }
      ...
    };
```

- The coroutine interface has to provide the API of an awaiter so that the interface can be used as an awaitable for `co_await`:

```
    class CoroTaskSub {
     public:
      ...
      bool await_ready() { return false; }    // do not skip suspension
      void await_suspend(auto awaitHdl) {
        awaitHdl.promise().subHdl = hdl;      // store sub-coroutine and suspend
      }
      void await_resume() { }
    };
```

- The coroutine interface has to resume the deepest sub-coroutine not done yet (if there is one):

```
  class CoroTaskSub {
   public:
    ...
    bool resume() const {
      if (!hdl || hdl.done()) {
        return false;          // nothing (more) to process
      }
      // find deepest sub-coroutine not done yet:
      CoroHdl innerHdl = hdl;
      while (innerHdl.promise().subHdl && !innerHdl.promise().subHdl.done()) {
        innerHdl = innerHdl.promise().subHdl;
      }
      innerHdl.resume();          // RESUME
      return !hdl.done();
    }
    ...
  };
```

coro/corotasksub.hpp provides the complete code.

With this, you can do the following:

coro/corocorosub.cpp

```
#include <iostream>
#include "corotasksub.hpp"    // for CoroTaskSub

CoroTaskSub coro()
{
  std::cout << "     coro(): PART1\n";
  co_await std::suspend_always{};     // SUSPEND
  std::cout << "     coro(): PART2\n";
}

CoroTaskSub callCoro()
{
  std::cout << "   callCoro(): CALL coro()\n";
  co_await coro();                    // call sub-coroutine
  std::cout << "   callCoro(): coro() done\n";
  co_await std::suspend_always{};     // SUSPEND
  std::cout << "   callCoro(): END\n";
}
```

```
int main()
{
  auto coroTask = callCoro();            // initialize coroutine
  std::cout << "MAIN: callCoro() initialized\n";

  while (coroTask.resume()) {            // RESUME
    std::cout << "MAIN: callCoro() suspended\n";
  }

  std::cout << "MAIN: callCoro() done\n";
}
```

Inside `callCoro()`, we can now call `coro()` by passing it to `co_await()`:

```
CoroTaskSub callCoro()
{
  ...
  co_await coro();                       // call sub-coroutine
  ...
}
```

What happens here is the following:

- The call of `coro()` initializes the coroutine and yields its coroutine interface of the type `CoroTaskSub`.
- Because this type has an awaiter interface, the coroutine interface can be used as an awaitable for `co_await`.
  The operator `co_await` is then called with two operands:
  - The awaiting coroutine where it is called
  - The coroutine that is called
- The usual behavior of an awaiter is applied:
  - `await_ready()` is asked whether to reject the request to await in general. The answer is “no” (`false`).
  - `await_suspend()` is called with the awaiting coroutine handle as parameter.
- By storing the handle of the sub-coroutine (the object that `await_suspend()` was called for) as the sub-coroutine of the passed awaiting handle, `callCoro()` now knows its sub-coroutine.
- By returning nothing in `await_suspend()`, the suspension is finally accepted, meaning that

  ```
  co_await coro();
  ```

  suspends `callCoro()` and transfers control flow back to the caller `main()`.
- When `main()` then resumes `callCoro()`, the implementation of `CoroTaskSub::resume()` finds `coro()` as its deepest sub-coroutine and resumes it.
- Whenever the sub-coroutine suspends, `CoroTaskSub::resume()` returns back to the caller.
- This happens until the sub-coroutine is done. The next resumptions will then resume `callCoro()`.

As a result, the program has the following output:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
MAIN: callCoro() suspended
    coro(): PART1
MAIN: callCoro() suspended
    coro(): PART2
MAIN: callCoro() suspended
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

Resuming Called Sub-Coroutines Directly

Note that the previous implementation of `CoroTaskSub` makes the call of

```
co_await coro();                    // call sub-coroutine
```

a suspend point. We initialize `coro()` but before we start it, we transfer the control flow back to the caller of `callCoro()`.

You could also start `coro()` here directly. To do this, you need only the following modification of `await_suspend()`:

```
auto await_suspend(auto awaitHdl) {
  awaitHdl.promise().subHdl = hdl;      // store sub-coroutine
  return hdl;                           // and resume it directly
}
```

If `await_suspend()` returns a coroutine handle, that coroutine is resumed immediately. This changes the behavior of our program to the following output:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
MAIN: callCoro() suspended
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

Returning another coroutine to resume can be used to resume any other coroutine on `co_await`, which we will use later for a symmetric transfer.

As you can see, the return type of `await_suspend()` can vary.

In any case, if `await_suspend()` signals that there should be no suspension, `await_resume()` will never be called.

14.4.4 Passing Values From Suspension Back to the Coroutine

Another application of awaitables and awaiters is to allow values to be passed back to the coroutine after suspension. Consider the following coroutine:

coro/coyieldback.hpp

```cpp
#include <iostream>
#include "corogenback.hpp"    // for CoroGenBack

CoroGenBack coro(int max)
{
  std::cout << "         CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "         CORO " << val << '/' << max << '\n';

    // yield next value:
    auto back = co_yield val;          // SUSPEND with value with response
    std::cout << "         CORO => " << back << "\n";
  }

  std::cout << "         CORO " << max << " end\n";
}
```

Again, the coroutine iterates up to a maximum value and on suspension, it yields the current value to the caller. However, this time, `co_yield` yields a value back from the caller to the coroutine:

```cpp
auto back = co_yield val;          // SUSPEND with value with response
```

To support that, the coroutine interface provides the usual API with some modifications:

coro/corogenback.hpp

```cpp
#include "backawaiter.hpp"
#include <coroutine>
#include <exception>    // for terminate()
#include <string>

class [[nodiscard]] CoroGenBack {
 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;                           // native coroutine handle

 public:
  struct promise_type {
    int coroValue = 0;                   // value TO caller on suspension
    std::string backValue;               // value back FROM caller after suspension

    auto yield_value(int val) {          // reaction to co_yield
      coroValue = val;                   // - store value locally
      backValue.clear();                 // - reinit back value
      return BackAwaiter<CoroHdl>{};     // - use special awaiter for response
    }

    // the usual members:
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

  // constructors and destructor:
  CoroGenBack(auto h) : hdl{h} { }
  ~CoroGenBack() { if (hdl) hdl.destroy(); }

  // no copying or moving:
  CoroGenBack(const CoroGenBack&) = delete;
  CoroGenBack& operator=(const CoroGenBack&) = delete;

  // API:
  // - resume the coroutine:
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;      // nothing (more) to process
    }
    hdl.resume();        // RESUME
    return !hdl.done();
  }

  // - yield value from co_yield:
  int getValue() const {
    return hdl.promise().coroValue;
  }

  // - set value back to the coroutine after suspension:
  void setBackValue(const auto& val) {
    hdl.promise().backValue = val;
  }
};
```

The modifications are:

- `promise_type` has a new member `backValue`.
- `yield_value()` returns a special awaiter of the type `BackAwaiter`.
- The coroutine interface has a new member `setBackValue()`, to send a value back to the coroutine.

Let us go through the changes in detail.

Promise Data Members

As usual, the promise type of the coroutine interface is the best place for data that the coroutine shares and exchanges with the caller:

```
class CoroGenBack {
  ...
 public:
  struct promise_type {
    int coroValue = 0;               // value TO caller on suspension
    std::string backValue;           // value back FROM caller after suspension
    ...
  };
  ...
};
```

We now have two data members there:

- `coroValue` for the value passed from the coroutine to the caller
- `backValue` for the value passed back from the caller to the coroutine

A New Awaiter for `co_yield`

To pass the value via the promise to the caller, we still have to implement `yield_value()`. However, this time, `yield_value()` does not return `std::suspend_always{}`. Instead, it returns a special awaiter of the type `BackAwaiter`, which will be able to yield the value returned from the caller:

```
class CoroGenBack {
  ...
 public:
  struct promise_type {
    int coroValue = 0;               // value TO caller on suspension
    std::string backValue;           // value back FROM caller after suspension

    auto yield_value(int val) {      // reaction to co_yield
      coroValue = val;               // - store value locally
      backValue.clear();             // - reinit back value
      return BackAwaiter<CoroHdl>{}; // - use special awaiter for response
    }
    ...
  };
  ...
};
```

The awaiter is defined as follows:

coro/backawaiter.hpp

```
template<typename Hdl>
class BackAwaiter {
  Hdl hdl = nullptr;     // coroutine handle saved from await_suspend() for await_resume()
 public:
  BackAwaiter() = default;

  bool await_ready() const noexcept {
    return false;                     // do suspend
  }

  void await_suspend(Hdl h) noexcept {
    hdl = h;                          // save handle to get access to its promise
  }

  auto await_resume() const noexcept {
    return hdl.promise().backValue;   // return back value stored in the promise
  }
};
```

All the awaiter does is to store the coroutine handle passed to `await_suspend()` locally so that we have it when `await_resume()` is called. In `await_resume()`, we use this handle to yield the `backValue` from its promise (stored by the caller with `setBackValue()`).

The return value of `yield_value()` is used as the value of the `co_yield` expression:

```
auto back = co_yield val;          // co_yield yields return value of yield_value()
```

The `BackAwaiter` is generic in that it could be used for all coroutine handles that have a member `backValue` of any arbitrary type.

The Coroutine Interface for the Value Sent Back after Suspension

As usual, we have to decide about the API that the coroutine interface provides to let the caller return a response. One simple approach is to provide the member function `setBackValue()` for that:

```
class CoroGenBack {
  ...
 public:
  struct promise_type {
    int coroValue = 0;               // value TO caller on suspension
    std::string backValue;           // value back FROM caller after suspension
    ...
  };
  ...
  // - set value back to the coroutine after suspension:
  void setBackValue(const auto& val) {
    hdl.promise().backValue = val;
  }
};
```

You could also provide an interface that yields a reference to `backValue` to support direct assignments to it.

Using the Coroutine

The coroutine can be used as follows:

coro/coyieldback.cpp

```
#include "coyieldback.hpp"
#include <iostream>
#include <vector>

int main()
{
  // start coroutine:
  auto coroGen = coro(3);                  // initialize coroutine
  std::cout << "**** coro() started\n";

  // loop to resume the coroutine until it is done:
  std::cout << "\n**** resume coro()\n";
  while (coroGen.resume()) {               // RESUME
    // process value from co_yield:
    auto val = coroGen.getValue();
    std::cout << "**** coro() suspended with " << val << '\n';

    // set response (the value co_yield yields):
    std::string back = (val % 2 != 0 ? "OK" : "ERR");
    std::cout << "\n**** resume coro() with back value: " << back << '\n';
    coroGen.setBackValue(back);            // set value back to the coroutine
  }

  std::cout << "**** coro() done\n";
}
```

This program has the following output:

```
**** coro() started

**** resume coro()
         CORO 3 start
         CORO 1/3
**** coro() suspended with 1
```

```
**** resume coro() with back value: OK
         CORO => OK
         CORO 2/3
**** coro() suspended with 2

**** resume coro() with back value: ERR
         CORO => ERR
         CORO 3/3
**** coro() suspended with 3

**** resume coro() with back value: OK
         CORO => OK
         CORO 3 end
**** coro() done
```

The coroutine interface has all the flexibility to handle things differently. For example, we could pass a response as a parameter of `resume()` or share the yielded value and its response in one member.

14.5 Afternotes

The request to support coroutines was first proposed as a pure library extension by Oliver Kowalke and Nat Goodspeed in `http://wg21.link/n3708`.

Due to the complexity of the feature, a *Coroutine TS* (experimental technical specification) was established with `http://wg21.link/n4403` to work on details.

The wording to finally merge coroutines into the C++20 standard was formulated by Gor Nishanov in `http://wg21.link/p0912r5` .

This page is intentionally left blank

Chapter 15

Coroutines in Detail

Now that we have learned about coroutines in the previous chapter, this chapter discusses several aspects of coroutines in detail.

15.1 Coroutine Constraints

Coroutines have the following attributes and constraints:

- Coroutines are not allowed to have `return` statements.
- Coroutines cannot be `constexpr` or `consteval`.
- Coroutines cannot have return type `auto` or other placeholder types.
- `main()` cannot be a coroutine.
- A constructor or destructor cannot be a coroutine.

A coroutine may be `static`. A coroutine may be a member function if it is not a constructor or destructor.

A coroutine may even be a lambda. However, in that case, care must be taken.

15.1.1 Coroutine Lambdas

A coroutine may be a lambda. We could also implement our first coroutine as follows:

```
auto coro = [] (int max) -> CoroTask {
              std::cout << "         CORO " << max << " start\n";

              for (int val = 1; val <= max; ++val) {
                // print next value:
                std::cout << "         CORO " << val << '/' << max << '\n';

                co_await std::suspend_always{};     // SUSPEND
              }
```

```
                 std::cout << "          CORO " << max << " end\n";
  };
```

However, note that a coroutine lambda ***should never capture anything***.[1] This is because a lambda is a shortcut for defining a function object that is created in the scope in which the lambda is defined. When you leave that scope, the coroutine lambda might still be resumed even though the lambda object has been destroyed.

Here is an example of **broken code** that results in a fatal runtime error:

```
auto getCoro()
{
  string str = "          CORO ";
  auto coro = [str] (int max) -> CoroTask {
                std::cout << str << max << " start\n";
                for (int val = 1; val <= max; ++val) {
                  std::cout << str << val << '/' << max << '\n';
                  co_await std::suspend_always{}; // SUSPEND
                }
                std::cout << str << max << " end\n";
              };
  return coro;
}

auto coroTask = getCoro()(3);      // initialize coroutine
// OOPS: lambda destroyed here
coroTask.resume();                 // FATAL RUNTIME ERROR
```

Normally, returning a lambda that captures something by value has no lifetime issues.

We could use the coroutine lambda as follows:

```
auto coro = getCoro();             // initialize coroutine lambda
auto coroTask = coro(3);           // initialize coroutine
coroTask.resume();                 // OK
```

We could also pass `str` to `getCoro()` by reference and ensure that `str` lives as long as we use the coroutine.

However, the fact that you cannot combine both initializations into one statement may be very surprising and more subtle lifetime issues can occur easily. Therefore, not using captures in coroutine lambdas at all is highly recommended.

[1] Thanks to Dietmar Kühl for pointing this out.

15.2 The Coroutine Frame and the Promises

When a coroutine is started, three things happen:

- A coroutine frame is created to store all necessary data of the coroutine. This usually happens on the heap.

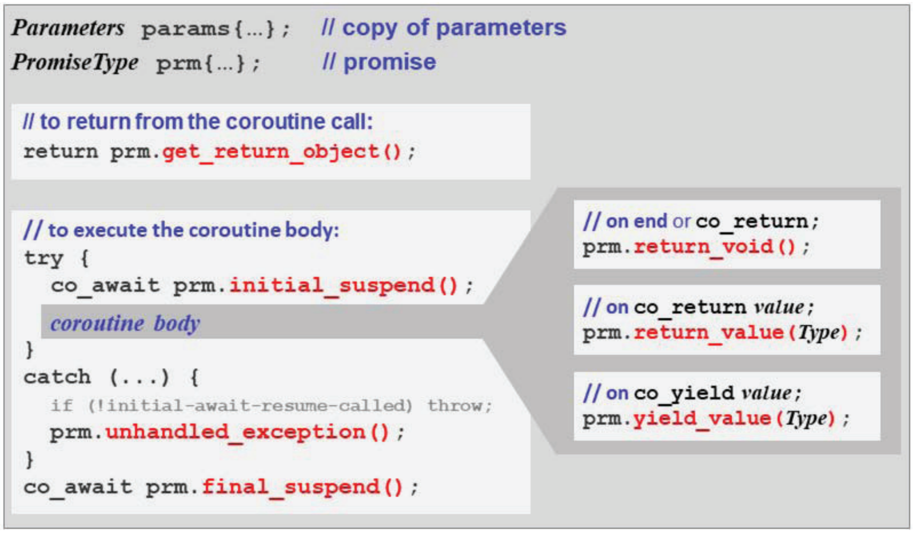
  However, compilers are allowed to put the coroutine frame on the stack. This typically happens if the lifetime of the coroutine is within the lifetime of the caller and the compiler has enough information to compute the size of the frame.
- All parameters of the coroutine are copied into the frame.

  Note that references are copied as references; it is not that their values are copied. This means that the arguments that the reference parameters refer to have to be valid as long as the coroutine is running.

  The advice is to never declare coroutine parameters as references. Otherwise, fatal runtime errors with undefined behavior may occur.
- The promise object is created inside the frame. Its purpose is to store the state the coroutine and provide hooks for customization while the coroutine is running.

  You can think of these objects as a "coroutine state controller" (an object that controls the behavior of the coroutine and can be used to track its state).[2]

Figure 15.1 visualizes this initialization and shows which customization points of the promise object are used while the coroutine is running.



```
Parameters  params{...};   // copy of parameters
PromiseType prm{...};      // promise

// to return from the coroutine call:
return prm.get_return_object();

// to execute the coroutine body:
try {
  co_await prm.initial_suspend();
  coroutine body
}
catch (...) {
  if (!initial-await-resume-called) throw;
  prm.unhandled_exception();
}
co_await prm.final_suspend();

// on end or co_return;
prm.return_void();

// on co_return value;
prm.return_value(Type);

// on co_yield value;
prm.yield_value(Type);
```

Figure 15.1. Coroutine frame and promise

[2] Thanks to Lewis Baker for pointing this out in
http://lewissbaker.github.io/2018/09/05/understanding-the-promise-type.

15.2.1 How Coroutine Interfaces, Promises, and Awaitables Interact

Let us also bring the awaitables into play again and see how work as a whole is organized:

- For each coroutine started, the compiler creates a promise.
- That promise, embedded in a coroutine handle, is then placed in a coroutine interface. That is, the coroutine interface usually controls the lifetime of the handle and its promise.
- On suspension, coroutines use awaitables to control what happens on suspension and resumption.

The following program demonstrates the exact control flow by using a tracing coroutine interface, promise, and awaiter. The tracing coroutine interface and promise are implemented as follows:

coro/tracingcoro.hpp

```
#include <iostream>
#include <coroutine>
#include <exception>   // for terminate()

// coroutine interface to deal with a simple task
// - providing resume() to resume it
class [[nodiscard]] TracingCoro {
 public:
  // native coroutine handle and its promise type:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
  CoroHdl hdl;             // coroutine handle

  // helper type for state and customization:
  struct promise_type {
    promise_type() {
      std::cout << "       PROMISE: constructor\n";
    }
    ~promise_type() {
      std::cout << "       PROMISE: destructor\n";
    }
    auto get_return_object() {         // init and return the coroutine interface
      std::cout << "       PROMISE: get_return_object()\n";
      return TracingCoro{CoroHdl::from_promise(*this)};
    }
    auto initial_suspend() {           // initial suspend point
      std::cout << "       PROMISE: initial_suspend()\n";
      return std::suspend_always{};   // - start lazily
    }
    void unhandled_exception() {       // deal with exceptions
      std::cout << "       PROMISE: unhandled_exception()\n";
      std::terminate();                // - terminate the program
    }
    void return_void() {                // deal with the end or co_return;
      std::cout << "       PROMISE: return_void()\n";
    }
    auto final_suspend() noexcept {   // final suspend point
      std::cout << "       PROMISE: final_suspend()\n";
      return std::suspend_always{};   // - suspend immediately
    }
  };

  // constructor and destructor:
  TracingCoro(auto h)
   : hdl{h} {              // store coroutine handle in interface
      std::cout << "         INTERFACE: construct\n";
  }
  ~TracingCoro() {
    std::cout << "         INTERFACE: destruct\n";
    if (hdl) {
      hdl.destroy();       // destroy coroutine handle
    }
  }
  // don't copy or move:
  TracingCoro(const TracingCoro&) = delete;
  TracingCoro& operator=(const TracingCoro&) = delete;

  // API to resume the coroutine
  // - returns whether there is still something to process
  bool resume() const {
    std::cout << "         INTERFACE: resume()\n";
    if (!hdl || hdl.done()) {
      return false;                     // nothing (more) to process
    }
    hdl.resume();                       // RESUME
    return !hdl.done();
  }
};
```

We trace:

- When the coroutine interface is initialized and destroyed
- When the coroutine interface resumes the coroutine
- Each promise operation

The tracing awaiter is implemented as follows:

coro/tracingawaiter.hpp

```
#include <iostream>

class TracingAwaiter {
  inline static int maxId = 0;
  int id;
 public:
  TracingAwaiter() : id{++maxId} {
    std::cout << "          AWAITER" << id << ": ==> constructor\n";
  }
  ~TracingAwaiter() {
    std::cout << "          AWAITER" << id << ": <== destructor\n";
  }
  // don't copy or move:
  TracingAwaiter(const TracingAwaiter&) = delete;
  TracingAwaiter& operator=(const TracingAwaiter&) = delete;

  //constexpr
  bool await_ready() const noexcept {
    std::cout << "          AWAITER" << id << ":     await_ready()\n";
    return false; // true: do NOT (try to) suspend
  }

  // Return type/value means:
  // - void: do suspend
  // - bool: true: do suspend
  // - handle: resume coro of the handle
  //constexpr
  bool await_suspend(auto) const noexcept {
    std::cout << "          AWAITER" << id << ":     await_suspend()\n";
    return false;
  }

  //constexpr
  void await_resume() const noexcept {
    std::cout << "          AWAITER" << id << ":     await_resume()\n";
  }
};
```

Here, we also trace each operation.

Note that the member functions cannot be `constexpr` because they have I/O.

The coroutine is implemented and used as follows:

coro/corotrace.cpp

```
#include "tracingcoro.hpp"
#include "tracingawaiter.hpp"
#include <iostream>

TracingCoro coro(int max)
{
  std::cout << "  START coro(" << max << ")\n";
  for (int i = 1; i <= max; ++i) {
    std::cout << "  CORO: " << i << '/' << max << '\n';
    co_await TracingAwaiter{};           // SUSPEND
    std::cout << "  CONTINUE coro(" << max << ")\n";
  }
  std::cout << "  END coro(" << max << ")\n";
}

int main()
{
  // start coroutine:
  std::cout << "**** start coro()\n";
  auto coroTask = coro(3);               // init coroutine
  std::cout << "**** coro() started\n";

  // loop to resume the coroutine until it is done:
  std::cout << "\n**** resume coro() in loop\n";
  while (coroTask.resume()) {            // RESUME
    std::cout << "**** coro() suspended\n";
    ...
    std::cout << "\n**** resume coro() in loop\n";
  }

  std::cout << "\n**** coro() loop done\n";
}
```

The program has the following output:

```
**** start coro()
      PROMISE: constructor
      PROMISE: get_return_object()
        INTERFACE: construct
      PROMISE: initial_suspend()
**** coro() started

**** resume coro() in loop
        INTERFACE: resume()
```

```
  START coro(3)
  CORO: 1/3
          AWAITER1: ==> constructor
          AWAITER1:     await_ready()
          AWAITER1:     await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER1:     await_resume()
          AWAITER1: <== destructor
  CONTINUE coro(3)
  CORO: 2/3
          AWAITER2: ==> constructor
          AWAITER2:     await_ready()
          AWAITER2:     await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER2:     await_resume()
          AWAITER2: <== destructor
  CONTINUE coro(3)
  CORO: 3/3
          AWAITER3: ==> constructor
          AWAITER3:     await_ready()
          AWAITER3:     await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER3:     await_resume()
          AWAITER3: <== destructor
  CONTINUE coro(3)
  END coro(3)
      PROMISE: return_void()
      PROMISE: final_suspend()

**** coro() loop done
        INTERFACE: destruct
      PROMISE: destructor
```

The first thing that happens when we call a coroutine is that the coroutine promise is created.

Then, `get_return_object()` is called for the promise created. This function usually initializes the coroutine handle and returns the coroutine interface initialized with it. To create the handle, usually the static member function `from_promise()` is called. The handle is then passed to initialize the coroutine interface of the type `TracingCoro`. The coroutine interface is then returned to the caller of `get_return_object()` so that it can be used as the return value of the call of the coroutine (in rare cases other return types are possible).

Then, `initial_suspend()` is called to see whether the coroutine should be immediately suspended (start lazily), which is the case here. Therefore, the control flow is given back to the caller of the coroutine.

Later, main() calls resume(), which calls resume() for the coroutine handle. This call resumes the coroutine, which means that the coroutine processes the following statements up to the next suspension or its end.

With each suspension, co_await uses an awaitable, which is an awaiter of the type TracingAwaiter in this case. It is created with the default constructor. For the awaiter, the member functions await_ready() and await_suspend() are called to control the suspension (they might even reject it). Because await_ready() returns false and await_suspend() returns nothing, the request to suspend is accepted. Therefore, the resumption of the coroutine ends and main() continues. On the next resumption, await_resume() is called and the coroutine continues.

When a coroutine reaches its end or a co_return statement, the corresponding member functions for dealing with its end are called. First, depending on whether or not there is a return value, return_void() or return_value() is called. Then, the function for the final suspension, final_suspend(), is called. Note that even inside final_suspend() the coroutine is still "running," which means that calling resume() again or destroy() inside final_suspend() would cause a runtime error.

At the end of the lifetime of the coroutine interface, its destructor is called, which destroys the coroutine handle. Finally, the promise is destroyed.

As an alternative, assume that initial_suspend() returns a promise type signaling to start the coroutine eagerly instead of initially suspending it:

```
class [[nodiscard]] TracingCoro {
 public:
  ...
  struct promise_type {
    ...
    auto initial_suspend() {         // initial suspend point
      std::cout << "      PROMISE: initial_suspend()\n";
      return std::suspend_never{};   // - start eagerly
    }
    ...
  };
  ...
};
```

In this case, we would get the following output:

```
**** start coro()
      PROMISE: constructor
      PROMISE: get_return_object()
        INTERFACE: construct
      PROMISE: initial_suspend()
  START coro(3)
  CORO: 1/3
          AWAITER1: ==> constructor
          AWAITER1:     await_ready()
          AWAITER1:     await_suspend()
**** coro() started

**** resume coro() in loop
        INTERFACE: resume()
```

```
          AWAITER1:     await_resume()
          AWAITER1: <== destructor
  CONTINUE coro(3)
  CORO: 2/3
          AWAITER2: ==> constructor
          AWAITER2:     await_ready()
          AWAITER2:     await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER2:     await_resume()
          AWAITER2: <== destructor
  CONTINUE coro(3)
  CORO: 3/3
          AWAITER3: ==> constructor
          AWAITER3:     await_ready()
          AWAITER3:     await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER3:     await_resume()
          AWAITER3: <== destructor
  CONTINUE coro(3)
  END coro(3)
      PROMISE: return_void()
      PROMISE: final_suspend()

**** coro() loop done
        INTERFACE: destruct
      PROMISE: destructor
```

Again, the coroutine framework creates the promise and calls `get_return_object()` for it, which initializes the coroutine handle and the coroutine interface. However, `initial_suspend()` does not suspend. Therefore, the coroutine starts eagerly executing the first statements until the first `co_await` suspends it for the first time. This suspension is the moment when the `TracingCoro` object that initially was created in `get_return_object()` is returned to the caller of the coroutine. After that, as before, we loop over the resumptions.

15.3 Coroutine Promises in Detail

Table *Special member functions of coroutine promises* lists all operations that are provided for coroutine handles in their promise type.

Operation	Effect
Constructor	Initializes the promise
`get_return_object()`	Defines the object that the coroutine returns (usually of the coroutine interface type)
`initial_suspend()`	Initial suspend point (to let the coroutine start lazily)
`yield_value(`*val*`)`	Deals with a value from `co_yield`
`unhandled_exception()`	Reaction to an unhandled exception inside the coroutine
`return_void()`	Deals with the end or a `co_return` that returns nothing
`return_value(`*val*`)`	Deals with a return value from `co_return`
`final_suspend()`	Final suspend point (to let the coroutine end lazily)
`await_transform(...)`	Maps values from `co_await` to awaiters
`operator new(`*sz*`)`	Defines the way the coroutine allocates memory
`operator delete(`*ptr*`, `*sz*`)`	Defines the way the coroutine frees memory
`get_return_object_on_...` `...allocation_failure()`	Defines the reaction if memory allocation failed

Table 15.1. Special member functions of coroutine promises

Some of these functions are mandatory, some of them depend on whether the coroutine yields intermediate or final results, and some of the are optional.

15.3.1 Mandatory Promise Operations

For coroutine promises, the following member functions are required. Without them, the code does not compile or runs into undefined behavior.

Constructor

The constructor of coroutine promises is called by the coroutine framework when a coroutine is initialized.

The constructor can be used by the compiler to initialize the coroutine state with some arguments. For this, the signature of the constructor has to match the arguments passed to the coroutine when it is called. This technique is especially used by coroutine traits.

`get_return_object()`

`get_return_object()` is used by the coroutine frame to create the return value of the coroutine.

The function usually has to return the coroutine interface, which is initialized with the coroutine handle, which is usually initialized with the static coroutine handle member function `from_promise()`, which itself is initialized with the promise.

For example:

```
class CoroTask {
 public:
  // native coroutine handle and its promise type:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
  CoroHdl hdl;

  struct promise_type {
    auto get_return_object() {         // init and return the coroutine interface
      return CoroTask{CoroHdl::from_promise(*this)};
    }
    ...
  }
  ...
};
```

In principle, `get_return_object()` may have different return types:

- The typical approach is that `get_return_object()` returns the coroutine interface by explicitly initializing it with the coroutine handle (as above).
- Instead, `get_return_object()` can also return the coroutine handle. In that case, the coroutine interface is implicitly created. This requires that the interface constructor is not `explicit`.

  It is currently not clear when exactly the coroutine interface is created in this case (see `http://wg21.link/cwg2563`). Therefore, the coroutine interface object might or might not exist when `initial_suspend()` is called, which can create trouble if something goes wrong.

  Therefore, you should better avoid to use this approach.
- In rare cases, it might even be useful to not return anything, and specify the return type to be `void`. Using coroutine traits can be one example.

`initial_suspend()`

`initial_suspend()` defines the *initial suspend point* of the coroutine, which is used primarily to specify whether the coroutine should automatically suspend after its initialization (start lazily) or start immediately (start eagerly).

The function is called for the promise `prm` of the coroutine as follows:

```
co_await prm.initial_suspend();
```

Therefore, `initial_suspend()` should return an awaiter.

Usually, `initial_suspend()` returns

- `std::suspend_always{}`, if the coroutine body is started later/lazily, or
- `std::suspend_never{}`, if the coroutine body is started immediately/eagerly

For example:

```
class CoroTask {
 public:
  ...
```

```
    struct promise_type {
      ...
      auto initial_suspend() {          // suspend immediately
        return std::suspend_always{};   // - yes, always
      }
      ...
    }
    ...
  };
```

However, you can also let the decision of an eager or lazy start depend on some business logic or use this function to perform some initialization for the scope of the coroutine body.

`final_suspend() noexcept`

`final_suspend()` defines the *final suspend point* of the coroutine and is called like `initial_suspend()` for the promise `prm` of the coroutine as follows:

```
  co_await prm.final_suspend();
```

This function is called by the coroutine frame outside the `try` block that encloses the coroutine body after `return_void()`, `return_value()`, or `unhandled_exception()` is called. Because `final_suspend()` is outside the `try` block, it has to be `noexcept`.

The name of this member function is a little misleading because it gives the impression that you could also return `std::suspend_never{}` here to force one more resumption after reaching the end of the coroutine. However, it is undefined behavior to resume a coroutine that is really suspended at the final suspend point. The only thing you can do with a coroutine suspended here is destroy it.

Therefore, the real purpose of this member function is to execute some logic, such as publishing a result, signaling completion, or resuming a continuation somewhere else.

Instead, the recommendation is to structure your coroutines so that they do suspend here where possible. One reason is that it makes it much easier for the compiler to determine when the lifetime of the coroutine frame is nested inside the caller of the coroutine, which makes it more likely that the compiler can elide heap memory allocation of the coroutine frame.

Therefore, unless you have a good reason not to do so, `final_suspend()` should always return `std::suspend_always{}`. For example:

```
  class CoroTask {
   public:
    ...
    struct promise_type {
      ...
      auto final_suspend() noexcept {   // suspend at the end
        ...
        return std::suspend_always{};   // - yes, always
      }
      ...
    }
    ...
  };
```

`unhandled_exception()`

`unhandled_exception()` defines the reaction when the coroutine body throws an exception. The function is called inside the `catch` clause of the coroutine frame.

Possible reactions to an exception here are:

- Ignoring the exception
- Processing the exception locally
- Ending or aborting the program (e.g., by calling `std::terminate()`)
- Storing the exception for later use with `std::current_exception()`

The way to implement these reactions is discussed later in a separate subsection.

In any case, after this function is called without ending the program, `final_suspend()` is called directly and the coroutine is suspended.

The coroutine is also suspended if you throw or rethrow an exception in `unhandled_exception()`. Any exception thrown from `unhandled_exception()` is ignored.

15.3.2 Promise Operations to Return or Yield Values

Depending on whether and how `co_yield` or `co_return` is used, you also need some of the following promise operations.

`return_void()` or `return_value()`

You have to implement exactly one the two member functions provided to deal with return statements and the end of the coroutine.:

- **`return_void()`**

  is called if the coroutine reaches the end (either by reaching the end of its body or by reaching a `co_return` statement without argument).

  See the first coroutine example for details.
- **`return_value(`*Type*`)`**

  is called if the coroutine reaches a `co_return` statement with an argument. The passed argument has to be or must convert to the specified type.

  See the coroutine example with `co_return` for details.

It is undefined behavior if coroutines are implemented in a way that they sometimes may and sometimes may not return a value. Consider this example:

```
ResultTask<int> coroUB(...)
{
  if (...) {
    co_return 42;
  }
}
```

This coroutine is not valid. Having both `return_void()` and `return_value()` is not permitted.[3]

Unfortunately, if you provide a `return_value(int)` member function only, this code might even compile and run and compilers might not even warn about it. This will hopefully change soon.

Note that you can ***overload*** `return_value()` for different types (except `void`) or make it generic:

```
struct promise_type {
  ...
  void return_value(int val) {           // reaction to co_yield for int
    ...                                  // - process returned int
  }
  void return_value(std::string val) {   // reaction to co_yield for string
    ...                                  // - process returned string
  }
};
```

In that case, a coroutine can `co_return` values of different types.

```
CoroGen coro()
{
  int value = 0;
  ...
  if (...) {
    co_return "ERROR: can't compute value";
  }
  ...
  co_return value;
}
```

`yield_value(`*Type*`)`

`yield_value(`*Type*`)` is called if the coroutine reaches a `co_yield` statement.

For basic details, see the coroutine example with `co_yield`.

You can overload `yield_value()` for different types or make it generic:

```
struct promise_type {
  ...
  auto yield_value(int val) {            // reaction to co_yield for int
    ...                                  // - process yielded int
    return std::suspend_always{};        // - suspend coroutine
  }
  auto yield_value(std::string val) {    // reaction to co_yield for string
    ...                                  // - process yielded string
    return std::suspend_always{};        // - suspend coroutine
  }
};
```

[3] There is an ongoing discussion about whether to allow promises to have both `return_void()` and `return_value()` in a future C++ standard.

In that case, a coroutine can `co_yield` values of different types:

```
CoroGen coro()
{
  while (...) {
    if (...) {
      co_yield "ERROR: can't compute value";
    }
    int value = 0;
    ...
    co_yield value;
  }
}
```

15.3.3 Optional Promise Operations

Promises can also be used to define some optional operations that define special behavior of coroutines, where normally some default behavior is used.

`await_transform()`

`await_transform()` can be defined to map values from `co_await` to awaiters.

`operator new()` and `operator delete()`

`operator new()` and `operator delete()` allow programmers to define a different way memory is allocated for the coroutine state.

These functions may also be used to ensure that coroutines do not accidentally use heap memory.

`get_return_object_on_allocation_failure()`

`get_return_object_on_allocation_failure()` allows programmers to define how to react to exceptionless failures of memory allocation for coroutines.

15.4 Coroutine Handles in Detail

The type `std::`**`coroutine_handle<>`** is *the* generic type for coroutine handles. It can be used to refer to an executing or suspended coroutine. Its template parameter is the promise type of the coroutine, which can be used to place additional data members and behavior.

Table *API of* `std::coroutine_handle<>` lists all operations that are provided for coroutine handles.

Operation	**Effect**
`coroutine_handle<`*PrmT*`>::from_promise(`*prm*`)`	Creates a handle with the promise *prm*
CoroHandleType`{}`	Creates a handle to no coroutine
CoroHandleType`{nullptr}`	Creates a handle to no coroutine
CoroHandleType`{`*hdl*`}`	Copies the handle *hdl* (both refer to the same coroutine)
hdl = *hdl2*	Assigns the handle *hdl2* (both refer to the same coroutine)
`if (`*hdl*`)`	Yields whether the handle refers to a coroutine
`==`, `!=`	Checks whether two handles refer to the same coroutine
`<`, `<=`, `>`, `>=`, `<=>`	Creates an order between coroutine handles
hdl`.resume()`	Resumes the coroutine
hdl`()`	Resumes the coroutine
hdl`.done()`	Yields whether a suspended coroutine is at its end and `resume()` is not allowed anymore
hdl`.destroy()`	Destroys the coroutine
hdl`.promise()`	Yields the promise of the coroutine
hdl`.address()`	Yields the internal address to the coroutine data
`coroutine_handle<`*PrmT*`>::from_address(`*addr*`)`	Yields the handle for the address *addr*

Table 15.2. API of `std::coroutine_handle<>`

The static member function `from_promise()` provides ***the*** way to initialize a coroutine handle with a coroutine. The function simply stores the address of the promise in the handle. If we have a promise *prm* this looks as follows:

```
auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
```

`from_promise()` is usually called in `get_return_object()` for a promise created by the coroutine framework:

```
class CoroIf {
 public:
  struct promise_type {
    auto get_return_object() {        // init and return the coroutine interface
      return CoroIf{std::coroutine_handle<promise_type>::from_promise(*this)};
```

```
        }
        ...
      };
      ...
    };
```

When the handle is default initialized or `nullptr` is used as the initial or assigned value, the coroutine handle does not refer to any coroutine. In that case, any conversion to a Boolean value yields `false`:

```
std::coroutine_handle<PrmType> hdl = nullptr;

if (hdl) ...          //false
```

Copying and assigning a coroutine handle is cheap. They just copy and assign the internal pointer. Therefore, coroutine handles are usually passed by value. This also means that multiple coroutine handles can refer to the same coroutine. As a programmer, you have to ensure that a handle never calls `resume()` or `destroy()` if another coroutine handle resumed or destroyed the coroutine.

The `address()` interface yields the internal pointer of the coroutine as `void*`. This allows programmers to export the coroutine handle to somewhere and later recreates a handle with the static function `from_address()`:

```
auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
...
void* hdlPtr = hdl.address();
...
auto hdl2 = std::coroutine_handle<decltype(prm)>::from_address(hdlPtr);
hdl == hdl2  //true
```

However, note that you can use the address only as long as the coroutine exists. After a coroutine has been destroyed, the address may be reused by another coroutine handle.

15.4.1 `std::coroutine_handle<void>`

All coroutine handle types have an implicit type conversion to class `std::coroutine<void>` (you can skip `void` in this declaration):[4]

```
namespace std {
  template<typename Promise>
  struct coroutine_handle {
    ...
    // implicit conversion to coroutine_handle<void>:
    constexpr operator coroutine_handle<>() const noexcept;
    ...
  };
}
```

[4] The original C++ standard specifies that all coroutine handle types ***derive*** from `std::coroutine<void>`. However, that was changed with `http://wg21.link/lwg3460`. Thanks to Yung-Hsiang Huang for pointing this out.

Thus, if you do not need the promise, you can always use the type `std::coroutine_handle<void>` or the type `std::coroutine_handle<>` without any template argument. Note that `std::coroutine_handle<>` does not provide the `promise()` member function:

```
void callResume(std::coroutine_handle<> h)
{
  h.resume();      // OK
  h.promise();     // ERROR: no member promise() provided for h
}

auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
...
callResume(hdl);   // OK: hdl converts to std::coroutine_handle<void>
```

15.5 Exceptions in Coroutines

When an exception is thrown inside a coroutine and this exception is not locally handled, `unhandled_exception()` is called in a `catch` clause behind the coroutine body (see Figure 15.1). It is also called in the case of exceptions from `initial_suspend()`, `yield_value()`, `return_void()`, `return_value()`, or any awaitable used within the coroutine.

You have the following options for dealing with these exceptions:

- Ignoring the exception

  In this case, `unhandled_exception()` just has an empty body:

  ```
  void unhandled_exception() {
  }
  ```

- Processing the exception locally

  In this case, `unhandled_exception()` just deals with the exception. Because you are inside a `catch` clause, you have to rethrow the exception and deal with it locally:

  ```
  void unhandled_exception() {
    try {
      throw;    // rethrow caught exception
    }
    catch (const std::exception& e) {
      std::cerr << "EXCEPTION: " << e.what() << std::endl;
    }
    catch (...) {
      std::cerr << "UNKNOWN EXCEPTION" << std::endl;
    }
  }
  ```

- Ending or aborting the program (e.g., by calling `std::terminate()`):

  ```
  void unhandled_exception() {
    ...
    std::terminate();
  }
  ```

- Storing the exception for later use with `std::current_exception()`
  In this case, you need an exception pointer in the promise type, which is then set:

```
struct promise_type {
  std::exception_ptr ePtr;
  ...
  void unhandled_exception() {
    ePtr = std::current_exception();
  }
};
```

You also have to provide a way to process the exceptions in the coroutine interface. For example:

```
class [[nodiscard]] CoroTask {
  ...
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;
    }
    hdl.promise().ePtr = nullptr;    // no exception yet
    hdl.resume();                    // RESUME
    if (hdl.promise().ePtr) {        // RETHROW any exception from the coroutine
      std::rethrow_exception(hdl.promise().ePtr);
    }
    return !hdl.done();
  }
};
```

Of course, combinations of these approaches are possible.

Note that after `unhandled_exception()` is called without ending the program, the coroutine is at its end. `final_suspend()` is called directly and the coroutine is suspended.

The coroutine is also suspended if you throw or rethrow an exception in `unhandled_exception()`. Any exception thrown from `unhandled_exception()` is ignored.

15.6 Allocating Memory for the Coroutine Frame

Coroutines need memory for their state. Because coroutines might switch contexts, heap memory is usually used. How this is done and how to change it is discussed in this section.

15.6.1 How Coroutines Allocate Memory

Coroutines need memory to store their state from suspensions to resumptions. However, because the resumptions may take place in very different contexts, in general, the memory can only be allocated on the heap. And in fact, the C++ standard states:

> An implementation may need to allocate additional storage for a coroutine. This storage is known as the coroutine state and is obtained by calling a non-array allocation function.

The important word is "may" here. Compilers are allowed to optimize away the need for heap memory. Of course, the compilers need enough information and the code has to be pretty simple. In fact, optimization is most likely if

- The lifetime of the coroutine remains within the lifetime of the caller.
- Inline functions are used so that the compiler can at least see everything to compute the size of the frame.
- `final_suspend()` returns `std::suspend_always{}` because otherwise, the lifetime management becomes too complicated.

However, at the time of writing this chapter (March 2022), only the Clang compiler was providing a corresponding *allocation elision* for coroutines.[5]

Consider, for example, the following coroutines:

```
CoroTask coro(int max)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "coro(" << max << "): " << val << '\n';
    co_await std::suspend_always{};
  }
}

CoroTask coroStr(int max, std::string s)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "coroStr(" << max << ", " << s << "): " << '\n';
    co_await std::suspend_always{};
  }
}
```

They differ in an additional string parameter for the second coroutine. Assume that we just pass some temporary objects to each parameter:

[5] Visual C++ was providing an optimization with the option `/await:heapelide`, but for the moment, it seems to be turned off.

```
  coro(3);              // create and destroy temporary coroutine
  coroStr(3, "hello");  // create and destroy temporary coroutine
```

If we have no optimization and track allocations, we might get something like the following output:

```
::new #1 (36 Bytes) => 0x8002ccb8
::delete (no size) at 0x8002ccb8

::new #2 (60 Bytes) => 0x8004cd28
::delete (no size) at 0x8004cd28
```

Here, the second coroutine needs 24 more bytes for the additional string parameter.

See *coro/coromem.cpp* for a complete example using *coro/tracknew.hpp* to see when heap memory is allocated or freed.

15.6.2 Avoiding Heap Memory Allocation

The promise type of a coroutine allow programmers to change the way memory is allocated for a coroutine. You simply have to provide the following members:

- `void* operator new(std::size_t sz)`
- `void operator delete(void* ptr, std::size_t sz)`

Ensure That no Heap Memory is Used

At first, the operators `new()` and `delete()` can be used to find out whether the coroutine allocated memory on the heap. Simply ***declare*** `operator new` without implementing it:[6]

```
  class CoroTask {
    ...
   public:
    struct promise_type {
      ...
      // find out whether heap memory is allocated:
      void* operator new(std::size_t sz);   // declared, but not implemented
    };
    ...
  };
```

Let Coroutines use no Heap Memory

Another use of these operators is to change the way the coroutine allocates memory. For example, you can map the calls for heap memory to some memory already allocated on the stack or in the data segment of the program.

Here is a concrete example of what this might look like:

[6] Thanks to Lewis Baker for pointing this out.

coro/corotaskpmr.hpp

```
#include <coroutine>
#include <exception>  // for terminate()
#include <cstddef>    // for std::byte
#include <array>
#include <memory_resource>

// coroutine interface to deal with a simple task
// - providing resume() to resume it
class [[nodiscard]] CoroTaskPmr {
  // provide 200k bytes as memory for all coroutines:
  inline static std::array<std::byte, 200'000> buf;
  inline static std::pmr::monotonic_buffer_resource
    monobuf{buf.data(), buf.size(), std::pmr::null_memory_resource()};
  inline static std::pmr::synchronized_pool_resource mempool{&monobuf};

 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;              // native coroutine handle

 public:
  struct promise_type {
    auto get_return_object() {         // init and return the coroutine interface
      return CoroTaskPmr{CoroHdl::from_promise(*this)};
    }
    auto initial_suspend() {           // initial suspend point
      return std::suspend_always{};    // - suspend immediately
    }
    void unhandled_exception() {       // deal with exceptions
      std::terminate();                // - terminate the program
    }
    void return_void() {               // deal with the end or co_return;
    }
    auto final_suspend() noexcept {    // final suspend point
      return std::suspend_always{};    // - suspend immediately
    }

    // define the way memory is allocated:
    void* operator new(std::size_t sz) {
      return mempool.allocate(sz);
    }
    void operator delete(void* ptr, std::size_t sz) {
      mempool.deallocate(ptr, sz);
    }
  };

  // constructor and destructor:
  CoroTaskPmr(auto h) : hdl{h} { }
  ~CoroTaskPmr() { if (hdl) hdl.destroy(); }

  // don't copy or move:
  CoroTaskPmr(const CoroTaskPmr&) = delete;
  CoroTaskPmr& operator=(const CoroTaskPmr&) = delete;

  // API to resume the coroutine
  // - returns whether there is still something to process
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;       // nothing (more) to process
    }
    hdl.resume();         // RESUME (blocks until suspended again or end)
    return !hdl.done();
  }
};
```

Here, we use ***P****olymorphic* ***M****emory* ***R****esources*, a feature introduced with C++17 that simplifies memory management by providing standardized memory pools. In this case, we pass a data segment of 200 kilobytes to the memory pool `monobuf`. Using the `null_memory_resource()` as fallback ensures that a `std::bad_alloc` exception is thrown if this memory is not enough. On top, we place the synchronized memory pool `mempool`, which manages this buffer with little fragmentation:[7]

```
// provide 200k bytes as memory for all coroutines:
std::array<std::byte, 200'000> buf;
std::pmr::monotonic_buffer_resource
  monobuf{buf.data(), buf.size(), std::pmr::null_memory_resource()};
std::pmr::synchronized_pool_resource mempool{&monobuf};
```

For simplicity, we create these objects as static inline members directly inside a coroutine interface `CoroTaskPmr` and provide `operator new` and `operator delete` to map "heap" memory requests for the coroutines to this memory pool:

```
class [[nodiscard]] CoroTaskPmr {
  // provide 200k bytes as memory for all coroutines:
  inline static std::array<std::byte, 200'000> buf;
  inline static std::pmr::monotonic_buffer_resource
    monobuf{buf.data(), buf.size(), std::pmr::null_memory_resource()};
```

[7] See my book *C++17 - The Complete Guide* for details.

```
    inline static std::pmr::synchronized_pool_resource mempool{&monobuf};
    ...

  public:
    struct promise_type {
      ...
      // define the way memory is allocated:
      void* operator new(std::size_t sz) {
        return mempool.allocate(sz);
      }
      void operator delete(void* ptr, std::size_t sz) {
        mempool.deallocate(ptr, sz);
      }
    };
    ...
};
```

We can use this coroutine interface as usual:

coro/coromempmr.cpp

```
#include <iostream>
#include <string>
#include "corotaskpmr.hpp"
#include "tracknew.hpp"

CoroTaskPmr coro(int max)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "    coro(" << max << "): " << val << '\n';
    co_await std::suspend_always{};
  }
}

CoroTaskPmr coroStr(int max, std::string s)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "    coroStr(" << max << ", " << s << "): " << '\n';
    co_await std::suspend_always{};
  }
}

int main()
{
  TrackNew::trace();
  TrackNew::reset();
  coro(3);                                // initialize temporary coroutine
  coroStr(3, "hello");                    // initialize temporary coroutine

  auto coroTask = coro(3);                // initialize coroutine
  std::cout << "coro() started\n";
  while (coroTask.resume()) {             // RESUME
    std::cout << "coro() suspended\n";
  }
  std::cout << "coro() done\n";
}
```

No more heap memory is allocated for these coroutines.

Note that you can also provide a specific `operator new` that takes the coroutine parameters after the size parameter. For example, for the coroutine that takes an `int` and a string, we could provide:

```
class CoroTaskPmr {
 public:
  struct promise_type {
    ...
    void* operator new(std::size_t sz, int, const std::string&) {
      return mempool.allocate(sz);
    }
  };
  ...
};
```

15.6.3 `get_return_object_on_allocation_failure()`

If the promise has a static member `get_return_object_on_allocation_failure()`, it is assumed that memory allocation never throws. By default, this has the effect that

```
::operator new(std::size_t sz, std::nothrow_t)
```

is called. In that case, the user-defined `operator new` must be `noexcept` and return `nullptr` on failure.

The function can then be used to implement workarounds, such as creating a coroutine handle that does not refer to a coroutine:

```
class CoroTask {
  ...
 public:
  struct promise_type {
    ...
    static auto get_return_object_on_allocation_failure() {
      return CoroTask{nullptr};
    }
  };
  ...
};
```

15.7 `co_await` and Awaiters in Detail

Now that we have introducing awaiters let us discuss them in detail and look up further examples.

15.7.1 Details of the Awaiter Interface

Table *Special member functions of awaiters* once again lists the key operations that awaiters have to provide.

Operation	Effect
Constructor	Initializes the awaiter
`await_ready()`	Yields whether suspension is (currently) disabled
`await_suspend(`*awaitHdl*`)`	Handle suspension
`await_resume()`	Handle resumption

Table 15.3. Special member functions of awaiters

Let us look at this awaiter interface in detail:

- *Constructor*

  The constructor allows coroutines (or wherever the awaiter is created) to pass arguments to the awaiter. These parameters can be used to impact or change the way the awaiter handles suspensions and resumptions. An awaiter for priority requests of a coroutine is one example.

- **`bool await_ready()`**

  is called right before the coroutine that this function is called for is suspended.

  It can be used to (temporarily) turn off suspension completely. If it returns `true`, we are "ready" to immediately return from the request to suspend and continue with the coroutine without suspending it.

  Usually, this function just returns `false` ("no, do not avoid/block any suspension"). However, it might conditionally yield `true` (e.g., if the suspension depends on some data being available).

  With its return type, `await_suspend()` can also signal not to accept a suspension of the coroutine (note that `true` and `false` have the opposite meaning there: by returning `true` in `await_suspend()`, the suspension is accepted). Not accepting a suspension with `await_ready()` has the benefit that the program saves the costs of initiating the suspension of the coroutine at all.

  Note that inside this function, the coroutine it is called for is not suspended yet. Do not (indirectly) call `resume()` or `destroy()` here. You can even call more complex business logic here as long as you can ensure that logic does not call `resume()` or `destroy()` for the coroutine that is suspended here.

- **`auto await_suspend(`*awaitHdl*`)`**

  is called right after the coroutine this function is called for is suspended. *awaitHdl* is the coroutine where the suspension is requested (with `co_await`). It has the type of the awaiting coroutine handle, `std::coroutine_handle<`*PromiseType*`>`.

  Here, you could specify what to do next, including resuming the suspended coroutine or the awaiting coroutine immediately and scheduling the other for later resumption. To support this, special return types may be used (this is discussed below).

  You could even destroy the coroutine here. However, in that case you have to ensure that the coroutine is not used anywhere else (such as calling `done()` in a coroutine interface).

- **`auto await_resume()`**
  is called when the coroutine this function is called for is resumed after a successful suspension (i.e., if `resume()` is called for the coroutine handle).
  The return type of `await_resume()` is the type of the value that the `co_await` or `co_yield` expression that caused the suspension yields. If it is not `void`, the context of the coroutine can return a value back to the resumed coroutine.

`await_suspend()` is the key function here. Its parameters and return values can vary as follows:

- The parameter of `await_suspend()` could be:
  - Using the type of the coroutine handle:
    `std::coroutine_handle<`*PrmType*`>`
  - Using a basic type usable for all coroutine handles:
    `std::coroutine_handle<void>` (or just `std::coroutine_handle<>`)
    In that case, you do not have access to the promise.
  - Using `auto` to let the compiler find out the type.
- The return type of `await_suspend()` could be:
  - `void` to continue with the suspension after performing the statements in `await_suspend()` and return back to the caller of the coroutine.
  - `bool` to signal whether the suspension should really happen. Here, `false` means "do ***not*** suspend (anymore)" (which is the opposite of the Boolean return values of `await_ready()`).
  - `std::coroutine_handle<>` to resume another coroutine instead.
    This use of `await_suspend()` is called ***symmetric transfer*** and is described later in detail.
    In this case, a ***noop coroutine*** can be used to signal not to resume any coroutine at all (in the same way as if the function were to return `false`).

In addition, note the following:

- The member functions are usually `const`, except when the awaiter has a member that is modified (such as storing the coroutine handle in `await_suspend()` to have it available on resumption).
- The member functions are usually `noexcept` (this is necessary to allow use in `final_suspend()`).
- The member functions can be `constexpr`.

15.7.2 Letting `co_await` Update Running Coroutines

Because awaiters can execute code on suspension, you can use them to change the behavior of a system that deals with coroutines. Let us look at an example where we let the coroutine change its priority.

Assume that we manage all coroutines in a scheduler and allow them to change their priorities with `co_await`. For that, we would first define a default priority and some arguments for `co_await` to change the priority:

```
int CoroPrioDefVal = 10;
enum class CoroPrioRequest {same, less, more, def};
```

With this, a coroutine might look as follows:

coro/coroprio.hpp

```
#include "corepriosched.hpp"
#include <iostream>

CoroPrioTask coro(int max)
{
  std::cout << "    coro(" << max << ")\n";
  for (int val = 1; val <= max; ++val) {
    std::cout << "    coro(" << max << "): " << val << '\n';
    co_await CoroPrio{CoroPrioRequest::less};    // SUSPEND with lower prio
  }
  std::cout << "    end coro(" << max << ")\n";
}
```

By using the special coroutine interface `CoroPrioTask`, the coroutine can use a special awaiter of the type `CoroPrio`, which allows the coroutine to pass a request for a priority change. On suspension, you can use it for the request to change the current priority:

```
co_await CoroPrio{CoroPrioRequest::less};    // SUSPEND with lower prio
```

For that, the main program creates a scheduler and calls `start()` to pass each coroutine to the scheduler:

coro/coroprio.cpp

```
#include "coroprio.hpp"
#include <iostream>

int main()
{
  std::cout << "start main()\n";
  CoroPrioScheduler sched;

  std::cout << "schedule coroutines\n";
  sched.start(coro(5));
  sched.start(coro(1));
  sched.start(coro(4));

  std::cout << "loop until all are processed\n";
  while (sched.resumeNext()) {
  }

  std::cout << "end main()\n";
}
```

The classes `CoroPrioScheduler` and `CoroPrioTask` interact as follows:

- The scheduler stores all coroutines in the order of their priority and provides members to store a new coroutine, resume the next coroutine, and change the priority of a coroutine:

```
class CoroPrioScheduler
{
  std::multimap<int, CoroPrioTask> tasks;   // all tasks sorted by priority
  ...
 public:
  void start(CoroPrioTask&& task);
  bool resumeNext();
  bool changePrio(CoroPrioTask::CoroHdl hdl, CoroPrioRequest pr);
};
```

- The member function `start()` of the scheduler stores the passed coroutine in the list and also stores the scheduler in the promise of each task:

```
class CoroPrioScheduler
{
  ...
 public:
  void start(CoroPrioTask&& task) {
    // store scheduler in coroutine state:
    task.hdl.promise().schedPtr = this;
    // schedule coroutine with a default priority:
    tasks.emplace(CoroPrioDefVal, std::move(task));
  }
  ...
};
```

- The class `CoroPrioTask` gives the class `CoroPrioScheduler` access to the handles, provides members in the promise type to point to the scheduler, and enables the coroutine to use `co_await` with a `CoroPrioRequest`:

```
class CoroPrioTask {
  ...
  friend class CoroPrioScheduler;              // give access to the handle
  struct promise_type {
    ...
    CoroPrioScheduler* schedPtr = nullptr;   // each task knows its scheduler:
    auto await_transform(CoroPrioRequest);   // deal with co_await a CoroPrioRequest
  };
  ...
}
```

For `start()`, it also provides move semantics the usual way.

The interface between the coroutine and the scheduler is the awaiter of the type CoroPrio. Its constructor takes the priority request and stores it in the promise on suspension (when we get the coroutine handle). For that, the constructor stores the request to have it available in await_suspend():

```
class CoroPrio {
 private:
  CoroPrioRequest prioRequest;
 public:
  CoroPrio(CoroPrioRequest pr)
   : prioRequest{pr} {    // deal with co_await a CoroPrioRequest
  }
  ...
  void await_suspend(CoroPrioTask::CoroHdl h) noexcept {
    h.promise().schedPtr->changePrio(h, prioRequest);
  }
  ...
};
```

The rest is just the usual boilerplate code for coroutine interfaces plus the handling of priorities. See *coro/coropriosched.hpp* for the complete code.

As shown, the main program schedules the three coroutines and loops until all have been processed:

```
sched.start(coro(5));
sched.start(coro(1));
sched.start(coro(4));

while (sched.resumeNext()) {
}
```

The output is as follows:

```
start main()
schedule coroutines
loop until all are processed
    coro(5)
    coro(5): 1
    coro(1)
    coro(1): 1
    coro(4)
    coro(4): 1
    coro(5): 2
    end coro(1)
    coro(4): 2
    coro(5): 3
    coro(4): 3
    coro(5): 4
    coro(4): 4
    coro(5): 5
    end coro(4)
    end coro(5)
end main()
```

Initially, the three coroutines started have the same priority. Because the priority of a coroutine is reduced each time it is suspended, the other coroutines then run in succession until the first coroutine has the highest priority again.

15.7.3 Symmetric Transfer with Awaiters for Continuation

The return type of `await_suspend()` can be a coroutine handle. In that case, a coroutine is suspended by immediately resuming the returned coroutine. This technique is called ***symmetric transfer***.

Symmetric transfer was introduced to improve performance and avoid stack overflows when coroutines call other coroutines. In general, when resuming a coroutine with `resume()`, the program needs a new stack frame for the new coroutine. If we were to resume a coroutine inside `await_suspend()` or after it was called, we would always pay that price. By returning the coroutine to be called, the stack frame for the current coroutine just replaces its coroutine so that no new stack frame is required.

Implementing Symmetric Transfer with Continuations

The typical application of this technique is implemented by using ***final awaiters*** that deal with ***continuations***.[8]

Assuming that you have a coroutine that ends and should then continue with another subsequent coroutine without giving control back to the caller, you run into the following problem: inside `final_suspend()`, your coroutine is ***not*** in a suspended state yet. You have to wait until `await_suspend()` is called for the returned awaitable to deal with any resumption. That might look as follows:

```
class CoroTask
{
 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;              // native coroutine handle
 public:
  struct promise_type {
    std::coroutine_handle<> contHdl = nullptr;   // continuation (if there is one)
    ...
    auto final_suspend() noexcept {
      // the coroutine is not suspended yet, use awaiter for continuation
      return FinalAwaiter{};
    }
  };
  ...
};
```

Thus, we return an awaiter that can be used to deal with the coroutine ***after*** it has been suspended. Here, the returned awaiter has the type `FinalAwaiter`, which might look as follows:

[8] This technique is documented with tremendous help from articles and emails by Lewis Baker and Michael Eiler. `http://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer` in particular provides a detailed motivation and explanation.

```
struct FinalAwaiter {
  bool await_ready() noexcept {
    return false;
  }
  std::coroutine_handle<> await_suspend(CoroTask::CoroHdl h) noexcept {
    // the coroutine is now suspended at the final suspend point
    // - resume its continuation if there is one
    if (h.promise().contHdl) {
      return h.promise().contHdl;       // return the next coro to resume
    }
    else {
      return std::noop_coroutine();    // no next coro => return to caller
    }
  }
  void await_resume() noexcept {
  }
};
```

Because await_suspend() returns a coroutine handle, the coroutine returned is automatically resumed on suspension. In that case, the utility function std::noop_coroutine() signals not to resume any other coroutine, meaning that the suspended coroutine returns to the caller.

std::noop_coroutine() returns a std::noop_coroutine_handle, which is an alias type of std::coroutine_handle<std::noop_coroutine_promise>. Coroutines of that type have no effect when calling resume() or destroy(), return nullptr when calling address(), and always return false when calling done().

std::noop_coroutine() and its return type are provided for situations in which await_suspend() may optionally return a coroutine to continue with. Having return type std::coroutine_handle<>, await_suspend() can return a noop coroutine to signal *not* to automatically resume another coroutine.

Note that two different return values of std::noop_coroutine() do not necessarily compare as equal. Therefore, the following code does not work portably:

```
std::coroutine_handle<> coro = std::noop_coroutine();
...
if (coro == std::noop_coroutine()) {   // OOPS: does not check whether coro has initial value
  ...
  return coro;
}
```

You should use nullptr instead:

```
std::coroutine_handle<> coro = nullptr;
...
if (coro) {                            // OK (checks whether coro has initial value)
  ...
  return std::noop_coroutine();
}
```

Dealing with coroutines in a thread pool demonstrates an application of this technique.

15.8 Other Ways of Dealing with `co_await`

So far, we have passed only awaiters to `co_await`. For example, by using a standard awaiter:

```
co_await std::suspend_always{};
```

Or, as another example, `co_await` took a user-defined awaiter:

```
co_await CoroPrio{CoroPrioRequest::less};    // SUSPEND with lower prio
```

However, `co_await` *expr* is an operator that can be part of a bigger expression and take values that do not have an awaiter type. For example:

```
co_await 42;
co_await (x + y);
```

The `co_await` operator has the same priority as `sizeof` or `new`. For this reason, you cannot skip the parentheses in the example above without changing the meaning. The statement

```
co_await x + y;
```

would be evaluated as follows:

```
(co_await x) + y;
```

Note that x + y or just x does not have to be an awaiter here. `co_await` needs an ***awaitable*** and *awaiters* are only one (typical) implementation of them. In fact, `co_await` accepts any value of any type provided there is a mapping to the API of an awaiter. For the mapping, the C++ standard provides two approaches:

- The promise member function `await_transform()`
- `operator co_await()`

Both approaches allow coroutines to pass any value of any type to `co_await` specifying an awaiter implicitly or indirectly.

15.8.1 `await_transform()`

If a `co_await` expression occurs in a coroutine, the compiler first looks up whether there is a member function `await_transform()` that is provided by the promise of the coroutine. If that is the case, `await_transform()` is called and has to yield an awaiter, which is then used to suspend the coroutine.

For example, this means that:

```
class CoroTask {
  struct promise_type {
    ...
    auto await_transform(int val) {
      return MyAwaiter{val};
    }
  };
  ...
};
```

```
CoroTask coro()
{
  co_await 42;
}
```

has the same effect as:

```
class CoroTask {
  ...
};

CoroTask coro()
{
  co_await MyAwaiter{42};
}
```

You can also use it to enable a coroutine to pass a value to the promise before using a (standard) awaiter:

```
class CoroTask {
  struct promise_type {
    ...
    auto await_transform(int val) {
      ...   // process val
      return std::suspend_always{};
    }
  };
  ...
};

CoroTask coro()
{
  co_await 42;   // let 42 be processed by the promise and suspend
}
```

Using Values to Let co_await Update Running Coroutines

Remember the example in which an awaiter was used to change the priority of the coroutine. There, we used an awaiter of the type CoroPrio to enable the coroutine to request the new priority as follows:

```
co_await CoroPrio{CoroPrioRequest::less};     // SUSPEND with lower prio
```

We could also allow only the new priority to be passed:

```
co_await CoroPrioRequest::less;                // SUSPEND with lower prio
```

All we would need is for the coroutine interface promise to provide a member function await_transform() for values of this type:

```
class CoroPrioTask {
 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
```

```
   private:
    CoroHdl hdl;                                // native coroutine handle
    friend class CoroPrioScheduler;             // give access to the handle
   public:
    struct promise_type {
      CoroPrioScheduler* schedPtr = nullptr;    // each task knows its scheduler:
      ...
      auto await_transform(CoroPrioRequest);    // deal with co_await CoroPrioRequest
    };
    ...
  };
```

The implementation of `await_transform()` may look as follows:

```
  inline auto CoroPrioTask::promise_type::await_transform(CoroPrioRequest pr) {
    auto hdl = CoroPrioTask::CoroHdl::from_promise(*this);
    schedPtr->changePrio(hdl, pr);
    return std::suspend_always{};
  }
```

Here, we again use the static member function `from_promise()` for coroutine handles to get the handle, because `changePrio()` needs the handle as its first argument.

That way, we can skip the awaiter `CoroPrio` as a whole. See *coro/corospriosched2.hpp* for the whole code (and *coro/coroprio2.cpp* and *coro/coroprio2.hpp* for its use).

Letting co_await Act Like co_yield

Remember the example with `co_yield`. To handle the value to be yielded we do the following:

```
  struct promise_type {
    int coroValue = 0;                  // last value from co_yield

    auto yield_value(int val) {         // reaction to co_yield
      coroValue = val;                  // - store value locally
      return std::suspend_always{};     // - suspend coroutine
    }
    ...
  };

  co_yield val;     // calls yield_value(val) on promise
```

We can get the same effect with the following:

```
  struct promise_type {
    int coroValue = 0;                  // last value from co_yield

    auto await_transform(int val) {
      coroValue = val;                  // - store value locally
      return std::suspend_always{};
    }
    ...
  };

  co_await val;     // calls await_transform(val) on promise
```

In fact,

```
  co_yield val;
```

is equivalent to

```
  co_await prm.yield_value(val);
```

where *prm* is the promise of the enclosing coroutine.

15.8.2 operator co_await()

The other option for letting `co_await` deal with any value of (almost) any type is to implement `operator co_await()` for that type. This means that you have to pass a value of a class.

Assume we have implemented `operator co_await()` for some type `MyType`:

```
  class MyType {
    auto operator co_await() {
      return std::suspend_always{};
    }
  };
```

Then, calling `co_await` for an object of that type:

```
  CoroTask coro()
  {
    ...
    co_await MyType{};
  }
```

calls the operator and uses the returned awaiter to handle the suspension. Here, `operator co_await()` yields `std::suspend_always{}`, which means that we effectively get:

```
  CoroTask coro()
  {
    ...
    co_await std::suspend_always{};
  }
```

However, you could pass arguments to `MyType{}` and pass them to an awaiter so that you can pass values to the suspended coroutine for immediate or later resumption.

Another example is to pass a coroutine to `co_await`, which can then be called, including implementing preparations such as changing threads or scheduling the coroutine a thread pool.

15.9 Concurrent Use of Coroutines

In principle, passing operands to `co_await` can execute any code. We can jump to a very different context or perform some operation and continue with the result of that operation as soon as we have it. Even when the other operation runs in a different thread, no synchronization mechanism is necessary.

This section provides a basic example to demonstrate this technique. Part of it is a simple thread pool to deal with coroutines, which uses `std::jthread` and a couple of new concurrency features of C++20.

15.9.1 `co_await` Coroutines

Assume we have the following coroutines calling each other:

coro/coroasync.hpp

```
#include "coropool.hpp"
#include <iostream>
#include <syncstream>    // for std::osyncstream

inline auto syncOut(std::ostream& strm = std::cout) {
  return std::osyncstream{strm};
}

CoroPoolTask print(std::string id, std::string msg)
{
  syncOut() << "    > " << id << " print: " << msg
            << "   on thread: " << std::this_thread::get_id() << std::endl;
  co_return;   // make it a coroutine
}

CoroPoolTask runAsync(std::string id)
{
  syncOut() << "===== " << id << " start     "
            << "   on thread: " << std::this_thread::get_id() << std::endl;

  co_await print(id + "a", "start");
  syncOut() << "===== " << id << " resume    "
            << "   on thread " << std::this_thread::get_id() << std::endl;

  co_await print(id + "b", "end  ");
  syncOut() << "===== " << id << " resume    "
            << "   on thread " << std::this_thread::get_id() << std::endl;

  syncOut() << "===== " << id << " done" << std::endl;
}
```

Both coroutines use `CoroPoolTask`, a coroutine interface for tasks running in a thread pool. The implementation of the interface and the pool will be discussed later.

The important part is that the coroutine `runAsync()` uses `co_await` to call another coroutine `print()`:

```
CoroPoolTask runAsync(std::string id)
{
  ...
  co_await print(...);
  ...
}
```

As we will see, this has the effect that the coroutine `print()` will be scheduled in a thread pool to run in different threads. In addition, `co_await` blocks until `print()` is done.

Note that `print()` needs a `co_return` to ensure that it is treated as a coroutine by the compiler. Without this, we would not have any `co_` keyword at all and the compiler (assuming this is an ordinary function) would complain that we have a return type but no return statement.

Note also that we use the helper `syncOut()`, which yields a std::osyncstream, to ensure that the concurrent output from different threads is synchronized line by line.

Assume we call the coroutine `runAsync()` as follows:

coro/coroasync1.cpp

```
#include "coroasync.hpp"
#include <iostream>

int main()
{
  // init pool of coroutine threads:
  syncOut() << "**** main() on thread " << std::this_thread::get_id()
            << std::endl;
  CoroPool pool{4};

  // start main coroutine and run it in coroutine pool:
  syncOut() << "runTask(runAsync(1))" << std::endl;
  CoroPoolTask t1 = runAsync("1");
  pool.runTask(std::move(t1));

  // wait until all coroutines are done:
  syncOut() << "\n**** waitUntilNoCoros()" << std::endl;
  pool.waitUntilNoCoros();

  syncOut() << "\n**** main() done" << std::endl;
}
```

Here, we first create a thread pool of the type `CoroPool` for all coroutines using the interface `CoroPoolTask`:

```
CoroPool pool{4};
```

Then, we call the coroutine, which starts the coroutine lazily and returns the interface, and hand the interface over to the pool to take control of the coroutine:

```
CoroPoolTask t1 = runAsync("1");
pool.runTask(std::move(t1));
```

We use (and have to use) move semantics for the coroutine interface, because `runTask()` requires to pass an rvalue to be able to take over the ownership of the coroutine (after the call `t1` no longer has it).

We could also do that in one statement:

```
pool.runTask(runAsync("1"));
```

Before we end the program. the pool blocks until all scheduled coroutines have been processed:

```
pool.waitUntilNoCoros();
```

When we run this program, we get something like the following output (the thread IDs vary):

```
**** main() on thread 0x80000008
runTask(runAsync(1))

**** waitUntilNoCoros()
===== 1 start        on thread: 0x8002cd90
    > 1a print: start   on thread: 0x8002ce68
===== 1 resume       on thread 0x8002ce68
    > 1b print: end     on thread: 0x8004d090
===== 1 resume       on thread 0x8004d090
===== 1 done

**** main() done
```

As you can see, different threads are used:

- The coroutine `runAsync()` is started on a different thread to `main()`
- The coroutine `print()` called from there is started on a third thread
- The second call of coroutine `print()` called from there is started on a fourth thread

However, `runAsync()` changes threads with each call of `print()`. By using `co_await`, these calls suspend `runAsync()` and call `print()` on a different thread (as we will see, the suspension schedules the called coroutine in the pool). At the end of `print()`, the calling coroutine `runAsync()` resumes on the same thread in which `print()` was running.

As a variation, we could also start and schedule coroutine `runAsync()` four times:

coro/coroasync2.cpp

```
#include "coroasync.hpp"
#include <iostream>

int main()
{
  // init pool of coroutine threads:
  syncOut() << "**** main() on thread " << std::this_thread::get_id()
            << std::endl;
  CoroPool pool{4};

  // start multiple coroutines and run them in coroutine pool:
  for (int i = 1; i <= 4; ++i) {
    syncOut() << "runTask(runAsync(" << i << "))" << std::endl;
    pool.runTask(runAsync(std::to_string(i)));
  }

  // wait until all coroutines are done:
  syncOut() << "\n**** waitUntilNoCoros()" << std::endl;
  pool.waitUntilNoCoros();

  syncOut() << "\n**** main() done" << std::endl;
}
```

This program might have the following output (using different thread IDs due to a different platform):

```
**** main() on thread 17308
runTask(runAsync(1))
runTask(runAsync(2))
runTask(runAsync(3))
runTask(runAsync(4))

**** waitUntilNoCoros()
===== 1 start        on thread: 18016
===== 2 start        on thread: 9004
===== 3 start        on thread: 17008
===== 4 start        on thread: 2816
    > 2a print: start    on thread: 2816
    > 1a print: start    on thread: 17008
===== 1 resume       on thread 17008
    > 4a print: start    on thread: 18016
===== 4 resume       on thread 18016
    > 3a print: start    on thread: 9004
===== 3 resume       on thread 9004
===== 2 resume       on thread 2816
    > 4b print: end      on thread: 9004
    > 1b print: end      on thread: 2816
===== 1 resume       on thread 2816
===== 1 done
===== 4 resume       on thread 9004
===== 4 done
```

```
    > 2b print: end     on thread: 17008
===== 2 resume       on thread 17008
===== 2 done
    > 3b print: end     on thread: 18016
===== 3 resume       on thread 18016
===== 3 done

**** main() done
```

The output now varies more because we have concurrent coroutines using the next available thread. However, in general, we have the same behavior:

- Whenever the coroutine calls `print()`, it is scheduled and probably started on a different thread (it might be the same thread if no other thread is available).
- Whenever `print()` is done, the calling coroutine `runAsync()` directly resumes on the same thread that `print()` was running on, meaning that `runAsync()` effectively changes threads with each call of `print()`.

15.9.2 A Thread Pool for Coroutine Tasks

Here is the implementation of the coroutine interface `CoroPoolTask` and the corresponding thread pool class `CoroPool`:

coro/coropool.hpp

```cpp
#include <iostream>
#include <list>
#include <utility>      // for std::exchange()
#include <functional>   // for std::function
#include <coroutine>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>

class CoroPool;

class [[nodiscard]] CoroPoolTask
{
  friend class CoroPool;
 public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
 private:
  CoroHdl hdl;
 public:
  struct promise_type {
    CoroPool* poolPtr = nullptr;     // if not null, lifetime is controlled by pool
    CoroHdl contHdl = nullptr;       // coro that awaits this coro

    CoroPoolTask get_return_object() noexcept {
      return CoroPoolTask{CoroHdl::from_promise(*this)};
    }
    auto initial_suspend() const noexcept { return std::suspend_always{}; }
    void unhandled_exception() noexcept { std::exit(1); }
    void return_void() noexcept { }

    auto final_suspend() const noexcept {
      struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }
        std::coroutine_handle<> await_suspend(CoroHdl h) noexcept {
          if (h.promise().contHdl) {
            return h.promise().contHdl;      // resume continuation
          }
          else {
            return std::noop_coroutine();    // no continuation
          }
        }
        void await_resume() noexcept { }
      };
      return FinalAwaiter{};  // AFTER suspended, resume continuation if there is one
    }
  };

  explicit CoroPoolTask(CoroHdl handle)
   : hdl{handle} {
  }
  ~CoroPoolTask() {
    if (hdl && !hdl.promise().poolPtr) {
      // task was not passed to pool:
      hdl.destroy();
    }
  }
  CoroPoolTask(const CoroPoolTask&) = delete;
  CoroPoolTask& operator= (const CoroPoolTask&) = delete;
  CoroPoolTask(CoroPoolTask&& t)
   : hdl{t.hdl} {
      t.hdl = nullptr;
  }
  CoroPoolTask& operator= (CoroPoolTask&&) = delete;

  // Awaiter for:  co_await task()
```

```
    // - queues the new coro in the pool
    // - sets the calling coro as continuation
    struct CoAwaitAwaiter {
      CoroHdl newHdl;
      bool await_ready() const noexcept { return false; }
      void await_suspend(CoroHdl awaitingHdl) noexcept;  // see below
      void await_resume() noexcept {}
    };
    auto operator co_await() noexcept {
      return CoAwaitAwaiter{std::exchange(hdl, nullptr)}; // pool takes ownership of hdl
    }
};

class CoroPool
{
 private:
  std::list<std::jthread> threads;          // list of threads
  std::list<CoroPoolTask::CoroHdl> coros;   // queue of scheduled coros
  std::mutex corosMx;
  std::condition_variable_any corosCV;
  std::atomic<int> numCoros = 0;            // counter for all coros owned by the pool

 public:
  explicit CoroPool(int num) {
    // start pool with num threads:
    for (int i = 0; i < num; ++i) {
      std::jthread worker_thread{[this](std::stop_token st) {
                                   threadLoop(st);
                                 }};
      threads.push_back(std::move(worker_thread));
    }
  }

  ~CoroPool() {
    for (auto& t : threads) {  // request stop for all threads
      t.request_stop();
    }
    for (auto& t : threads) {  // wait for end of all threads
      t.join();
    }
    for (auto& c : coros) {    // destroy remaining coros
      c.destroy();
    }
  }
```

```
CoroPool(CoroPool&) = delete;
CoroPool& operator=(CoroPool&) = delete;

void runTask(CoroPoolTask&& coroTask) noexcept {
  auto hdl = std::exchange(coroTask.hdl, nullptr); // pool takes ownership of hdl
  if (coroTask.hdl.done()) {
    coroTask.hdl.destroy();  // OOPS, a done() coroutine was passed
  }
  else {
    schedule coroutine in the pool
  }
}

// runCoro(): let pool run (and control lifetime of) coroutine
// called from:
// - pool.runTask(CoroPoolTask)
// - co_await task()
void runCoro(CoroPoolTask::CoroHdl coro) noexcept {
  ++numCoros;
  coro.promise().poolPtr = this;  // disables destroy in CoroPoolTask
  {
    std::scoped_lock lock(corosMx);
    coros.push_front(coro);       // queue coro
    corosCV.notify_one();         // and let one thread resume it
  }
}

void threadLoop(std::stop_token st) {
  while (!st.stop_requested()) {
    // get next coro task from the queue:
    CoroPoolTask::CoroHdl coro;
    {
      std::unique_lock lock(corosMx);
      if (!corosCV.wait(lock, st, [&] {
                                     return !coros.empty();
                                   })) {
        return;  // stop requested
      }
      coro = coros.back();
      coros.pop_back();
    }

    // resume it:
    coro.resume();  // RESUME
```

```
      // NOTE: The coro initially resumed on this thread might NOT be the coro finally called.
      //       If a main coro awaits a sub coro, then the thread that finally resumed the sub coro
      //       resumes the main coro as its continuation.
      //       => After this resumption, this coro and SOME continuations MIGHT be done
      std::function<void(CoroPoolTask::CoroHdl)> destroyDone;
      destroyDone = [&destroyDone, this](auto hdl) {
                      if (hdl && hdl.done()) {
                        auto nextHdl = hdl.promise().contHdl;
                        hdl.destroy();          // destroy handle done
                        --numCoros;             // adjust total number of coros
                        destroyDone(nextHdl);   // do it for all continuations done
                      }
                    };
      destroyDone(coro);         // recursively destroy coroutines done
      numCoros.notify_all();     // wake up any waiting waitUntilNoCoros()

      // sleep a little to force another thread to be used next:
      std::this_thread::sleep_for(std::chrono::milliseconds{100});
    }
  }

  void waitUntilNoCoros() {
    int num = numCoros.load();
    while (num > 0) {
      numCoros.wait(num);   // wait for notification that numCoros changed the value
      num = numCoros.load();
    }
  }
};

// CoroPoolTask awaiter for:  co_await task()
// - queues the new coro in the pool
// - sets the calling coro as continuation
void CoroPoolTask::CoAwaitAwaiter::await_suspend(CoroHdl awaitingHdl) noexcept
{
  newHdl.promise().contHdl = awaitingHdl;
  awaitingHdl.promise().poolPtr->runCoro(newHdl);
}
```

Both classes `CoroPoolTask` and `CoroPool` carefully play together:

- `CoroPoolTask` is used to initialize a coroutine as task to call. The task should be schedule in the pool, which then takes control over it until it gets destroyed.
- `CoroPool` implements a minimal thread pool that can run scheduled coroutines. In addition, it provides a very minimal API to
  - block until all scheduled coroutines are done
  - shutdown the pool (which is automatically done by its destructor)

Let us look at the code in detail.

Class CoroPoolTask

Class CoroPoolTask provides a coroutine interface to run a task with the following specialties:

- Each coroutine knows has a pointer to the thread pool, which is initialized when the thread pool takes control over the coroutine:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct promise_type {
    CoroPool* poolPtr = nullptr;     // if not null, lifetime is controlled by pool
    ...
  };
  ...
};
```

- Each coroutine has a member for an optional continuation as described before:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct promise_type {
    ...
    CoroHdl contHdl = nullptr;       // coro that awaits this coro
    ...
    auto final_suspend() const noexcept {
      struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }
        std::coroutine_handle<> await_suspend(CoroHdl h) noexcept {
          if (h.promise().contHdl) {
            return h.promise().contHdl;      // resume continuation
          }
          else {
            return std::noop_coroutine();   // no continuation
          }
        }
        void await_resume() noexcept {}
      };
      return FinalAwaiter{};   // AFTER suspended, resume continuation if there is one
    }
  };
  ...
};
```

- Each coroutine has an `operator co_await()`, which means that a `co_await task()` uses a special awaiter so that the `co_await` suspends the current coroutine after scheduling the passed coroutine to the pool with the current coroutine as continuation.

Here is how the implemented `operator co_await()` works. This is the relevant code:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct CoAwaitAwaiter {
    CoroHdl newHdl;
    bool await_ready() const noexcept { return false; }
    void await_suspend(CoroHdl awaitingHdl) noexcept;   // see below
    void await_resume() noexcept {}
  };
  auto operator co_await() noexcept {
    return CoAwaitAwaiter{std::exchange(hdl, nullptr)}; // pool takes ownership of hdl
  }
};

void CoroPoolTask::CoAwaitAwaiter::await_suspend(CoroHdl awaitingHdl) noexcept
{
  newHdl.promise().contHdl = awaitingHdl;
  awaitingHdl.promise().poolPtr->runCoro(newHdl);
}
```

When we call `print()` with `co_await`:

```
CoroPoolTask runAsync(std::string id)
{
  ...
  co_await print(...);
  ...
}
```

the following happens:

- `print()` is called as a coroutine and initialized.
- For the returned coroutine interface of the type `CoroPoolTask`, `operator co_await()` is called.
- `operator co_await()` takes the handle to initialize an awaiter of the type `CoAwaitAwaiter`. This means that the new coroutine `print()` becomes the member `newHdl` of the awaiter.
- `std::exchange()` ensures that in the initialized coroutine interface, the handle `hdl` becomes `nullptr` so that a destructor does not call `destroy()`.
- `runAsync()` is suspended with the awaiter initialized by `operator co_await()`, which calls `await_suspend()` with the handle of the coroutine `runAsync()` as parameter.
- Inside `await_suspend()` we store the passed suspended `runAsync()` as a continuation and schedule the `newHdl` (`print()`) in the pool to be resumed by one of the threads.

The Class `CoroPool`

`CoroPool` implements its minimal thread pool by a combination of some features introduced here and in other parts of this book.

First let us look at the members:

- Like any typical thread pool, this class has a member for the `threads` using the new thread class `std::jthread` so that we do not have to deal with exceptions and can signal a stop:

  ```
  std::list<std::jthread> threads;          // list of threads
  ```

  We do not need any synchronization because the only other moment we use `threads` after initialization is during a `shutdown()`, which first joins all threads (after signaling stop). Copying and moving the pool is not supported (this is for simplicity; we could easily provide move support).

  Note that for better performance, the destructor first requests all threads to stop before starting to `join()` them.

- Then, we have a queue of coroutines to process with corresponding synchronization members:

  ```
  std::list<CoroPoolTask::CoroHdl> coros;   // queue of scheduled coros
  std::mutex corosMx;
  std::condition_variable_any corosCV;
  ```

- To avoid stopping the pool too early, the pool also tracks the total number of coroutines owned by the pool:

  ```
  std::atomic<int> numCoros = 0;            // counter for all coros owned by the pool
  ```

  and provides the member function `waitUntilNoCoros()`.

The member function `runCoro()` is the key function for scheduling a coroutine for resumption. It takes a coroutine handle of the interface `CoroPoolTask` as an argument. There are two approaches for scheduling the coroutine interface itself:

- Calling `runTask()`
- Using `co_await task()`

Both approaches ***move*** the coroutine handle into the pool, so that the pool is then has the responsibility to `destroy()` the handle when it is no longer in use.

However, implementing the right moment to call `destroy()` (and adjust the total number of coroutines) is not easy to do correctly and safely. The coroutine should be finally suspended for that, which rules out doing this in `final_suspend()` of the task or `await_suspend()` of the final awaiter. So, tracking and destruction works as follows:

- Each time a coroutine handle is passed to the pool, we increment the number of coroutines:

  ```
    void runCoro(CoroPoolTask::CoroHdl coro) noexcept {
      ++numCoros;
      ...
    }
  ```

- After any resumption by a thread is done, we check whether the coroutine and possible continuations have been done. Note that the continuations are called automatically by the coroutine frame according to `await_suspend()` of the final awaiter of the task.

Therefore, after each resumption has finished, we recursively iterate over all continuations to destroy all of the coroutines done:

```
std::function<void(CoroPoolTask::CoroHdl)> destroyDone;
destroyDone = [&destroyDone, this](auto hdl) {
                if (hdl && hdl.done()) {
                  auto nextHdl = hdl.promise().contHdl;
                  hdl.destroy();            // destroy handle done
                  --numCoros;               // adjust total number of coros
                  destroyDone(nextHdl);     // do it for all continuations done
                }
              };
destroyDone(coro);          // recursively destroy coroutines done
```

Because the lambda is used recursively, we have to forward declare it as a `std::function<>`.

- Finally, we wake up any waiting `waitUntilNoCoros()` with the new thread synchronization feature of atomic types:

```
    numCoros.notify_all();    // wake up any waiting waitUntilNoCoros()
  ...
  void waitUntilNoCoros() {
    int num = numCoros.load();
    while (num > 0) {
      numCoros.wait(num);    // wait for notification that numCoros changed the value
      num = numCoros.load();
    }
  }
```

- If the pool is destroyed, we also destroy all remaining coroutine handles after all threads have finished.

Synchronous Waiting for Asynchronous Coroutines

In real-world coroutine pools, a lot of aspects might look different, more sophisticated, and use additional tricks to make the code more robust and safe.

For example, pools might provide a way to schedule a task and wait for its end, which for `CoroPool`, would look as follows:[9]

```
class CoroPool
{
  ...
  void syncWait(CoroPoolTask&& task) {
    std::binary_semaphore taskDone{0};
    auto makeWaitingTask = [&]() -> CoroPoolTask {
      co_await task;
      struct SignalDone {
        std::binary_semaphore& taskDoneRef;
```

[9] Thanks to Lewis Baker for pointing this out.

```
          bool await_ready() { return false; }
          bool await_suspend(std::coroutine_handle<>) {
            taskDoneRef.release();   // signal task is done
            return false;            // do not suspend at all
          }
          void await_resume() { }
        };
        co_await SignalDone{taskDone};
      };
      runTask(makeWaitingTask());
      taskDone.acquire();
    }
  };
```

Here we use a binary semaphore so that we can signal the end of the passed `task` and wait for it in a different thread.

Note that you have to be careful about what exactly is signaled. Here, the signal says we are behind the `task` that called with `co_await` was performed. In this case, you could even signal this in `await_ready()`:

```
          bool await_ready() {
            taskDoneRef.release();   // signal task is done
            return true;             // do not suspend at all
          }
```

In general, you have to consider that in `await_ready()`, the coroutine is not suspended yet. Therefore, any signal that causes it to check whether it is `done()` or even `destroy()` it results in a fatal runtime error. Because we use concurrent code here, you have to ensure that the signal does not even cause corresponding calls indirectly.

15.9.3 What C++ Libraries Will Provide After C++20

All you have seen here in this section of the coroutine chapter is just a pretty simple code example that is not robust, thread-safe, complete, or flexible enough to provide general solutions for various typical use cases of coroutines.

The C++ standards committee is working on better solutions to provide them in future libraries.

For a program that can run coroutines concurrently in a safe and flexible way, you need at least the following components:[10]

- A kind of task type that allows you to chain coroutines together
- Something that allows you to start multiple coroutines that run independently and join them later
- Something like `syncWait()` that allows a synchronous function to block waiting for an asynchronous function
- Something that allows multiple coroutines to be multiplexed on a smaller number of threads

[10] Thanks to Lewis Baker for pointing this out.

The Class CoroPool combines the last three aspects together more or less. However, the more flexible approach is to come with individual building blocks that can be combined in various ways. In addition, good thread-safety needs much better design and implementation techniques.

We are still learning and discussing how to do that best. Therefore, even in C++23, there will probably only be minimal support (if any).

15.10 Coroutine Traits

So far, in all examples, the coroutine interface was returned by the coroutine. However, you can also implement coroutines so that they use an interface created somewhere else. Consider the following example (a modified version of the first coroutine example):

```
void coro(int max, CoroTask&)
{
  std::cout << "CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "CORO " << val << '/' << max << '\n';

    co_await std::suspend_always{};      // SUSPEND
  }

  std::cout << "CORO " << max << " end\n";
}
```

Here, the coroutine takes the coroutine interface of the type `CoroTask` as a parameter. However, we have to specify that the parameter is used as the coroutine interface. This is done by a specialization of the template `std::coroutine_traits<>`. Its template parameters have to specify the signature (return type and parameter types) and inside, we have to map the type member `promise_type` to the type member of the coroutine interface parameter:

```
template<>
struct std::coroutine_traits<void, int, CoroTask&>
{
  using promise_type = CoroTask::promise_type;
};
```

The only thing we now have to ensure is that the coroutine interface can be created without a coroutine and refer to a coroutine later. For this, we need a promise with a constructor that takes the same parameters as the coroutine

```
class CoroTask {
 public:
  // required type for customization:
  struct promise_type {
    promise_type(int, CoroTask& ct) {   // init passed coroutine interface
```

```
        ct.hdl = CoroHdl::from_promise(*this);
      }
      void get_return_object() {           // nothing to do anymore
      }
      ...
    };

  private:
    // handle to allocate state (can be private):
    using CoroHdl = std::coroutine_handle<promise_type>;
    CoroHdl hdl;

  public:
    // constructor and destructor:
    CoroTask() : hdl{} {                   // enable coroutine interface without handle
    }
    ...
  };
```

Now the code using the coroutine might look as follows:

```
  CoroTask coroTask;   // create coroutine interface without coroutine

  // start coroutine:
  coro(3, coroTask);   // init and store coroutine in the interface created

  // loop to resume the coroutine until it is done:
  while (coroTask.resume()) {    // resume
    std::this_thread::sleep_for(500ms);
  }
```

You can find the complete example in *coro/corotraits.cpp*.

The promise constructor gets all parameters passed to the coroutine. However, because the promise usually only needs the coroutine interface, you might pass the interface as the first parameter and then use `auto&&...` as the last parameter.

This page is intentionally left blank

Chapter 16

Modules

This chapter presents the new C++20 feature *modules*. Modules provide a way to combine code from multiple files into one logical entity (module, component). In the same way as for classes, data encapsulation helps to provide a clear definition of the API of the module. As a side effect, it is possible to ensure that module code does not have to be compiled multiple times, even if it is placed in "header files."

This chapter was written with tremendous help and support from Daniela Engert and Hendrik Niemeyer, who also gave great introductions to this topic at the Meeting C++ conference 2020 and the ACCU 2021 conference, respectively.

16.1 Motivation for Modules Using a First Example

Modules allow programmers to define an API for code in the large. The code might consist of multiple classes, multiple files, several functions, and various auxiliary utilities including templates. By using the keyword `export`, you can specify what is exported as the API of a module that wraps all code that provides a certain functionality. Thus, we can define a clean API of a component implemented in various files.

Let us look at a few simple examples that declare a module in one file and then use this module in another file.

16.1.1 Implementing and Exporting a Module

The specification of the API of a module is defined in its ***primary interface*** (the official name is ***primary module interface unit***), which each module has exactly once:

modules/mod0.cppm

```
export module Square;   // declare module Square

int square(int i);

export class Square {
 private:
    int value;
  public:
    Square(int i)
     : value{square(i)} {
    }
    int getValue() const {
      return value;
    }
};

export template<typename T>
Square toSquare(const T& x) {
  return Square{x};
}

int square(int i) {
  return i * i;
}
```

The first thing you might recognize is that the file uses a new file extension: **.cppm**. The file extension of module files is not clear yet. We will discuss the handling of module files by compilers later.

The key entry for the primary interface is the line that declares and exports the module using the name Square:

```
export module Square;    // declare module Square
```

Note that the name is used only as an identifier to import the module. It does ***not*** introduce a new scope or namespace. Any name exported by the module is still in the scope it was in when it was exported.

The name of a module may contain periods, While periods are not valid in any other kinds of identifiers used in C++, a period as a character of a module name identifier is valid and has no special meaning. For example:

```
export module Math.Square;    // declare module "Math.Square"
```

This has no other effect than naming the module with "MathDotSquare," except that using a period has a visual effect. Periods can be used to signal some logical relationships between modules established by a component or project. Using them has no syntactic or formal consequences.

The public API of a module is defined by everything that is explicitly exported using the keyword export. In this case, we export the class Square and the function template toSquare<>():

```
export class Square {
  ...
};

export template<typename T>
Square toSquare(const T& x) {
   ...
}
```

Everything else is ***not*** exported and cannot be used directly by code that imports the module (we will discuss later how non-exported module symbols may be reachable but not visible). Therefore, the function `square()` declared without `export` cannot be used by code that imports this module.

The file looks like a header file with the following differences:

- We have the line with the module declaration.
- We have symbols, types, functions (even templates) that are exported with `export`.
- We do not need `inline` to define functions.
- We do not need preprocessor guards.

However, the module file is not just an improved header file. A module file can play the role of both a header and a source file. It can contain declarations and definitions. Furthermore, in a module file, you do not have to specify a definition with `inline` or within preprocessor guards. Entities exported by modules cannot violate the ***One Definition Rule*** when imported by different translation units.

Each module must have exactly one *primary interface* file with its specified name. As you can see, the name of the module does ***not*** conflict with any symbol *within* the module. It also does not implicitly introduce a namespace. Therefore, a module can have the name of its (major) namespace, class, or function. In practice, module names may often fit with the namespace of the symbols exported. This, however, is something you have to implement explicitly.

16.1.2 Compiling Module Units

As you can see, a module file may have both declarations and definitions. In the traditional sense, it can be seen as a combination of header and source file. The consequence is that you have to do two things with it:

- **Precompile the declarations** (including all generic code), which converts the declarations into a compiler-specific format
- **Compile the definition**, which creates the usual object files

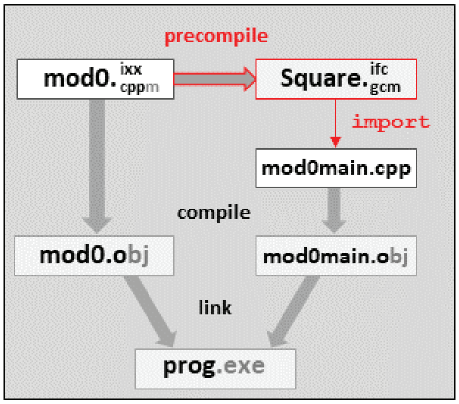


Figure 16.1. Dealing with C++ modules

Given that we have the primary module interface **mod0.cppm** above, we have to deal with it in two ways as Figure 16.1 demonstrates:

- We have to precompile mod0.cppm to create a precompiled module file that contains all exported declarations including precompiled template definitions. It is identified by the name of the module Square, not by the name of the source file.
- We have to compile mod0.cppm to create an object file mod0.o or mod0.obj with the assembler code of all definitions that can be compiled directly.

As already stated, there are no specific file extensions required for source module files. I use .cppm here. There is also no standardized suffix for precompiled module files. It is up to the compiler to decide about them. By default, we currently have the following situation:

- gcc/g++ uses .gcm for precompiled files (and places them in a subdirectory gcm.cache).
- Visual C++ uses .ifc for precompiled files (and places them in the local directory).

We will discuss file suffixes and options for dealing with module units later in detail.

Note that a successful compilation of a source file that imports modules requires that the precompiled artifact of the module is available. Thus, you have to precompile mod0.cppm before you can compile mod0test.cpp. If you do not follow the correct order, you might import a version of the specified module that is not up to date. As a consequence, cyclic import dependencies are not allowed.

In contrast to other programming languages, C++ does not require a module to have a special file name or be in a special directory. Any C++ file can define a module (but only one) and the name of the module need not have anything to do with the name or location of the file.

Of course, it makes good sense to keep file names and module names synchronized somehow. However, that decision ultimately depends on your preferences and the constraints of the configuration management and build systems that you use.

16.1.3 Importing and Using a Module

To use the code of a module in a program, you have to import the module under its name. Here is a simple example of a program using just the module Square defined above:

modules/mod0main.cpp

```
#include <iostream>

import Square;   // import module "Square"

int main()
{
  Square x = toSquare(42);
  std::cout << x.getValue() << '\n';
}
```

With

```
import Square;    // import module "Square"
```

we import all exported symbols from the module `Square`. This means that we can then use the exported class `Square` and the function template `toSquare<>()`.

Using any symbol from the module that is not exported results in a compile-time error:

```
import Square;    // import module "Square"

square(42)        // ERROR: square() not exported
```

Again, note that a module does *not* automatically introduce a new namespace. We use the exported symbols of the module in the scope they were in when they were exported. If you want to have everything exported from the module in its own namespace, you can exporting the namespace as a whole.

16.1.4 Reachable versus Visible

When using modules, a new distinction comes into play: *reachability* versus *visibility*.[1] When exporting data, we might not be able to see and directly use a name or symbol of the module; although we might be able to use it indirectly.

Symbols that are reachable but not visible can occur when an exported API provides access to a type that is not exported. Consider the following example:

```
export module ModReach;    // declare module ModReach

struct Data {              // declare a type not exported
  int value;
};

export struct Customer {   // declare an exported type
 private:
  Data data;
 public:
  Customer(int i)
   : data{i} {
  }
  Data getData() const {   // yield a type not exported
    return data;
  }
};
```

[1] Thanks to Daniela Engert for pointing this out.

When importing this module, type Data is ***not visible*** and therefore cannot be used directly:

```
import ModReach;
...

Data d{11};                       // ERROR: type Data not exported
Customer c{42};
const Data& dr = c.getData();     // ERROR: type Data not exported
```

However, type Data is ***reachable*** and that thus be used indirectly:

```
import ModReach;
...

Customer c{42};
const auto& dr = c.getData();     // OK: type Data is used
auto d = c.getData();             // OK: d has type Data
std::cout << d.value << '\n';     // OK: type Data is used
```

You can even declare an object of type Data as follows:

```
decltype(std::declval<Customer>().getData()) d;    // d has non-exported type Data
```

By using `std::declval<>()`, we call `getData()` for an assumed object of type `Customer`. Therefore, we declare `d` with the type `Data`, the return type of `getData()`, if called for an object of type `Customer`.

Private module fragments can be used to restrict the reachability of indirectly exported classes and functions.

The visibility and reachability of exported symbols are discussed later in detail.

16.1.5 Modules and Namespaces

As already mentioned, symbols of a module are imported in the same scope as when they were exported. In contrast to some other programming languages, C++ modules do ***not*** automatically introduce a namespace for a module.

You might therefore use the convention of exporting everything in a module in a namespace with its name. You can do that in two ways:

- Specify the component(s) you want to export with `export` within the namespace:

```
export module Square;    // declare module "Square"

namespace Square {
  int square(int i);

  export class Square {
    ...
  };
```

```
    export template<typename T>
    Square toSquare(const T& x) {
      ...
    }

    int square(int i) {   // not exported
      ...
    }
  }
```

• Specify everything you want to export in a namespace declared with `export`:

```
  export module Square;   // declare module "Square"

  int square(int i);

  export namespace Square {
    class Square {
      ...
    };

    template<typename T>
    Square toSquare(const T& x) {
      ...
    }
  }

  int square(int i) {   // not exported
    ...
  }
```

In both cases, the module exports class `Square::Square` and `Square::toSquare<>()` (therefore, the namespace of the symbols is exported, even if not marked with `export`).

Using the module would now look as follows:

```
#include <iostream>

import Square;   // import module "Square"

int main()
{
  Square::Square x = Square::toSquare(42);
  std::cout << v.getValue() << '\n';
}
```

16.2 Modules with Multiple Files

The purpose of modules is to deal with code of significant size distributed over multiple files. Modules can be used to wrap code of small, medium-sized, and very large components consisting of 2, 10, or even 100 files. These files might even be provided and maintained by multiple programmers and teams.

To demonstrate the scalability of this approach and its benefits, let us now look at how multiple files can be used to define a module that can be used/imported by other code. The code size of the example is still small so that you would normally not spread it over multiple files. The goal is to demonstrate the features with very simple examples.

16.2.1 Module Units

In general, modules consist of multiple ***module units***. Module units are translation units that belong to a module.

All module units have to be compiled in some way. Even if they contain only declarations, which would go in header files in traditional code, some kind of precompilation is necessary. Therefore, the files are always transferred into some internal platform-specific format that is used to avoid having to (pre)compile the same code again and again.

Besides the ***primary module interface unit***, C++ provides three other unit types to split the code of a module into multiple files:

- ***Module implementation units*** allow programmers to implement definitions in their own file so that they can be compiled separately (similar to traditional C++ source code in `.cpp` files).
- ***Internal partitions*** allow programmers to provide declarations and definitions that are visible only within a module in separate files.
- ***Interface partitions*** even allow programmers to split the exported module API into multiple files.

The next sections introduce these additional module units with examples.

16.2.2 Using Implementation Units

The first example of a module that is implemented in multiple files demonstrates how to split *definitions* (such as function implementations) to avoid having them in one file. The usual motivation for this is to be able to compile the definitions separately.

This can be done by using ***module implementations*** (the official name is ***module implementation units***). They are handled like traditional source files that are compiled separately.

Let us look at an example.

A Primary Interface with a Global Module Fragment

As usual, first we need ***the*** primary interface that defines what we export:

modules/mod1/mod1.cppm

```
module;                    // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod1;   // module declaration

struct Order {
  int count;
  std::string name;
  double price;

  Order(int c, const std::string& n, double p)
   : count{c}, name{n}, price{p} {
  }
};

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(const std::string& n)
   : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

This time, the module starts with `module;` to signal that we have a module. That way we can use some preprocessor commands within modules:

```
module;                    // start module unit with global module fragment

#include <iostream>
#include <string>
#include <vector>

export module Mod1;        // module declaration
...
```

The area between `module;` and the module declaration is called the ***global module fragment***. You can use it to place preprocessor commands like `#define` and `#include`. Nothing within this area is exported (no macros, no declarations, no definitions).

Nothing else can be done before we formally start the module unit with its declaration (except comments, of course):

```
export module mod1;        // module declaration
```

The things defined in this module are:

- An internal data structure `Order`:

  ```
  struct Order {
    ...
  };
  ```

  This data structure is for order entries. Each entry holds information about how many items were ordered, their name, and their price. The constructor ensures that we initialize all members.
- A class `customer`, which we export:

  ```
  export class Customer {
    ...
  };
  ```

As you can see, we need the header files and the internal data structure `Order` to define the class `Customer`. However, by not exporting them, they cannot be used directly by code that imports this module.

For the class `Customer`, the member functions `averagePrice()`, `sumPrice()`, and `print()` are only declared. Here, we use the feature to define them in ***module implementation units***.

Module Implementation Units

A module may have any number of implementation units. In our example, we provide two of them: one to implement numeric operations and one to implement I/O operations.

The module implementation unit for numeric operations looks as follows:

modules/mod1/mod1price.cpp

```
module Mod1;              // implementation unit of module Mod1

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  if (orders.empty()) {
    return 0.0;
  }
  return sumPrice() / orders.size();
}
```

The file is a module implementation unit because it starts with a declaration stating that this is a file of module `Mod1`:

```
module Mod1;
```

This declaration imports the primary interface unit of the module (but nothing else). Thus, the declarations of types `Order` and `Customer` are known and we can provide implementations of their member functions directly.

Note that a module implementation unit does ***not*** `export` anything. `export` is allowed only in interface files of modules (primary interface or interface partition), which are declared with `export module` (and remember that only one primary interface is allowed per module).

Again, a module implementation unit may start with a global module fragment, which we can see in the module implementation unit for I/O:

modules/mod1/mod1io.cpp

```
module;                   // start module unit with global module fragment

#include <iostream>
#include <format>

module Mod1;              // implementation unit of module Mod1

void Customer::print() const
{
  // print name:
```

```
  std::cout << name << ":\n";
  // print order entries:
  for (const auto& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                             od.count, od.name, od.price, od.count * od.price);
  }
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", "    Sum:", sumPrice());
}
```

Here, we introduce the module with `module;` to have a global module fragment for the header files that we use in the implementation unit. `<format>` is the header file of the new formatting library.

As you can see, module implementation units use the file extension of traditional C++ translation units (most of the time `.cpp`). Compilers deal with them just like any other non-module C++ code.

Using the Module

The code using the module looks as follows:

modules/mod1/testmod1.cpp

```
#include <iostream>

import Mod1;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);

  c1.print();
  std::cout << "  Average: " << c1.averagePrice() << '\n';
}
```

Here, we use the exported class `Customer` from the primary interface to create a customer, place some orders, print the customer with all orders, and print the value of the average order.

The program has the following output:

```
Kim:
  1 table           59.90  59.90
  4 chair            9.20  36.80
                          ------
    Sum:                   96.70
  Average: 48.35
```

Note that any attempt to use type `Order` in the code that imports the module results in a compile-time error.

Note also that the use of the module does not depend on how many implementation units we have. The number of implementation units matters only in that the linker has to use all object files generated for them.

16.2.3 Internal Partitions

In the previous example, we introduced a data structure `Order` that is only used within the module. It looks like we have to declare it in the primary interface to make it available to all implementation units, which of course does not really scale in large projects.

With ***internal partitions***, you can declare and define internal types and functions of a module in separate files. Note that *partitions* can also be used to define parts of an exported interface in a separate file, which we will discuss later.

Note that *internal partitions* are sometimes called *partition implementation units*, which is based on the fact that in the C++20 standard, they are officially called "*module implementation units* that are *module partitions*" and that sounds like they provide the implementations of interface partitions. They do not. They just act like internal header files for a module and may provide both declarations and definitions.

Defining an Internal Partition

Using an internal partition, we can define the local type `Order` in its own module unit as follows:

modules/mod2/mod2order.cppp

```
module;                     // start module unit with global module fragment

#include <string>

module Mod2:Order;          // internal partition declaration

struct Order {
  int count;
  std::string name;
  double price;

  Order(int c, const std::string& n, double p)
   : count{c}, name{n}, price{p} {
  }
};
```

As you can see, a partition has the name of the module, then a colon, and then its partition name:

```
module Mod2:Order;
```

Sub-partitions such as `Mod2:Order:Main` are not supported.

You might again recognize that the file uses another new file extension: **`.cppp`**, which we will discuss later after looking at its contents.

The primary interface has to import this partition by only using the name `:Order`:

modules/mod2/mod2.cppm

```
module;                 // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod2;     // module declaration

import :Order;          // import internal partition Order

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(const std::string& n)
   : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
};
```

The primary interface has to import the internal partition because it uses type `Order`. With that import, the partition is available in all units of the module. If the primary interface does not need type `Order` and does not import the internal partition, all module units that need type `Order` would have to import the internal partition directly.

Again, note that partitions are only an internal implementation aspect of a module. For the user of the code, it does not matter whether code is in the primary module, its implementation, or in an internal partition. However, code from internal partitions cannot be exported.

16.2.4 Interface Partitions

You can also split the interface of a module into multiple files. In that case, you declare ***interface partitions***, which themselves export whatever should be exported.

Interface partitions are especially useful if modules provide multiple interfaces that are maintained by different programmers and/or teams. For simplicity, let us just use the current example to demonstrate how to use this feature by defining only the `Customer` interface in a separate file.

To define only the `Customer` interface, we can provide the following file:

modules/mod3/mod3customer.cppm

```
module;                        // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod3:Customer;   // interface partition declaration

import :Order;                 // import internal partition to use Order

export class Customer {
 private:
  std::string name;
  std::vector<Order> orders;
 public:
  Customer(const std::string& n)
   : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

The partition is more or less the former primary interface with one difference:

- As a partition, we declare its name after the module name and a colon: `Mod3:Customer`

Like the primary interface:

- We export this module partition:

  ```
  export module Mod3:Customer;
  ```

- We use the new file extension **.cppm**, which we will again discuss later

The primary interface is still the only place to specify what a module exports. However, the primary module can delegate exports to the interface partition. The way to do that is to export the imported interface partition directly as a whole:

modules/mod3/mod3.cppm

```
export module Mod3;         // module declaration

export import :Customer;    // import and export interface partition Customer
...                         // import and export other interface partitions
```

By importing the interface partition and exporting it at the same time (yes, you have to write both keywords), the primary interface exports the interface of the partition `Customer` as its own interface:

```
export import :Customer;   // import and export partition Customer
```

Importing an interface partition without exporting it is not allowed.

Again, note that partitions are only an internal implementation aspect of a module. It does not matter whether interfaces and implementations are provided in partitions. Partitions do not create a new scope.

Therefore, for the implementation of the member functions of `Customer`, moving the declaration of the class to a partition does not matter. You implement a member function of the class `Customer` as part of the module `Mod3`:

modules/mod3/mod3io.cpp

```
module;                    // start module unit with global module fragment

#include <iostream>
#include <vector>
#include <format>

module Mod3;               // implementation unit of module Mod3

import :Order;             // import internal partition to use Order

void Customer::print() const
{
  // print name:
  std::cout << name << ":\n";
  // print order entries:
  for (const Order& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                             od.count, od.name, od.price, od.count * od.price);
  }
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", "    Sum:", sumPrice());
}
```

However, there is one difference in this implementation unit: because the primary interface no longer imports the internal partition `:Order`, this module has to do so because it uses type `Order`.

For code that imports the module, the way the code is distributed internally also does not matter. We still export the class `Customer` in the global scope:

modules/mod3/testmod3.cpp

```cpp
#include <iostream>

import Mod3;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);

  c1.print();
  std::cout << "  Average: " << c1.averagePrice() << '\n';
}
```

16.2.5 Summary of Splitting Modules into Different Files

The examples in this section demonstrate how you can deal with modules of increasing code size so that splitting the code is helpful or even necessary to "tame the beast:"

- **Module implementation units** allow projects to split definitions into multiple files so that the source code can be maintained by different programmers and you do not have to recompile all code if local things change.
- **Internal partitions** allow projects to move module-local declarations and definitions outside the primary interface. They can be imported by the primary interface or only by the module units that need them.
- **Interface partitions** allow projects to maintain exported interfaces in different files. This usually makes sense if the exported API becomes so big that it helps to have different files (and therefore teams) to work on parts of it.

The **primary interface** brings everything together and specifies what is exported to the users of the module (by directly exporting symbols or exporting imported interface partitions).

The kind of module unit we have depends on the `module` declaration within a C++ source file (which can come after comments and a global module fragment for preprocessor commands):

- **`export module`** *name*`;`
  identifies *the* primary interface. For each module, it may exist only once in a C++ program.
- **`module`** *name*`;`
  identifies an implementation unit providing just a definition (which might use local declarations). You can have as many of them as you like.

- **`module`** *name*`:`*partname*`;`
  identifies an internal partition with declarations and definitions used only within the module. You can have multiple partitions, but for each *partname*, you can have only one internal partition file.
- **`export module`** *name*`:`*partname*`;`
  identifies an interface partition. You can have multiple interface partitions, but for each *partname*, you can have only one interface partition file.

Because we have no standard suffixes for the different module units, tools have to parse the beginning of the C++ source files to detect whether and which kind of a module unit they are. Note that the module declaration might occur after comments and the global module fragment. See **clmod.py** in `http://github.com/josuttis/cppmodules` for a Python script that demonstrates how this might look.

16.3 Dealing with Modules in Practice

This section discusses some additional aspects of modules for using them in practice.

16.3.1 Dealing with Module Files with Different Compilers

In C++, extensions are not standardized. In practice, different file extensions are used (usually `.cpp` and `.hpp`, but also `.cc`, `.cxx`, `.C`, `.hh`, `.hxx`, `.H`, and even `.h` are used).

We also have no standard extension for modules. Even worse, we do not even agree on whether new extensions are necessary (yet). In principle, there are two approaches:

- Compilers should consider all kinds of module files as ordinary C++ source files and find out how to handle them based on their content. With this approach, all files still have the extension `.cpp`. gcc/g++ follows this policy.
- Compilers treat (some) module files differently as they can be both files for declarations (traditional header files) and files with definitions (traditional source files). While there is no doubt that it helps tremendously to have different suffixes for that reason, compilers might even indirectly require different suffixes to avoid having to use different command-line options for the same extensions. Visual C++ follows this approach.

Therefore, different file extensions are recommended for module files in practice by different compilers (`.cppm`, `.ixx`, and `.cpp`), which is one reason why using modules in practice is still challenging.

I thought about this for a while and tried things out and discussed the situation with the people in the standards committee, but so far, there seems to be no compelling solution. Formally, the C++ standard does not standardize the way source code is handled (the code might not even be stored in files). This has the consequence that there is no easy way to write even a simple portable first example using modules.

Therefore, let me suggest something here so that you can at least try modules out on different platforms:

There are multiple reasons why different file extensions seem to be necessary:

- Compilers require different command-line options for dealing with different types of module files.
- Similar to header files, you have to provide ***some*** of the module files to customers and third-party code.
- Different module files create different artifacts, which you might have to deal with (e.g., when removing the generated artifacts).

Personally, I have no ***final*** decision and recommendation for file extensions of module files yet. However, given the current situation, I recommend the following:

- For **interface files** (both primary interfaces and interface partitions), use the file extension **`.cppm`**. The reasons are:
  - It is the best self-explanatory file extension (far better than `.ixx` as currently recommended by Visual C++).
  - This is what Clang currently requires.
  - It can be used by gcc.
  - Visual C++ requires special treatment anyway, unless you use the extension `.ixx`.
- For **module implementation files** (but ***not*** partition implementation files), use the usual file extension **`.cpp`**. The reasons are:
  - No special artifacts are generated.
  - No special command-line options are required.
- For **internal partition files** (partition implementation files), use the file extension **`.cppp`**. The reasons are:
  - Visual C++ requires the command-line option `/internalPartition` for these files. The file suffix does not matter. Therefore, you have to use a special suffix to have a chance of a generic rule in build systems that do not want to parse the file contents.
  - It can be used by gcc.
  - Currently, Clang does not support these files at all (September 2021).

  The way Microsoft handles internal partitions is really counter-productive for the success of modules and I hope they will fix the need for a specific suffix as soon as possible.

As a consequence, you have to (pre)compile module files as follows:

- **Visual C++:**

  Visual C++ requires specific command-line extensions and prefers a different file extension `.ixx` than the one I propose. For this reason:
  - Compile an interface file `file.cppm` as follows:

    ```
    cl /TP /interface /c file.cppm
    ```

    The option `/TP` specifies that all following files contain C++ source code. Alternatively, you can use `/Tpfile.cppm`. The option `/interface` specifies that ***all*** following files are interface files (having both interface and non-interface files on one command-line might not work).

    If you use the file extension **`.ixx`** instead, the compiler recognizes the file automatically as an interface file.
  - Compile an internal partition file `file.cppp` as follows:

    ```
    cl /Tp /internalPartition /c file.cppp
    ```

    The option `/internalPartition` specifies that ***all*** following files are internal partitions. Note that having both internal partition and interface files on one command-line is not supported. There is no specific suffix you can use instead; internal partitions always need this option.

  The fact that Visual C++ currently recommends different file suffixes and requires specific command-line options for specific module units makes the use of modules both cumbersome and non-portable. To circumvent the restrictions of Visual C++ (at least when you compile via command line), I have provided the Python script **clmod.py**, which you can find at `http://github.com/josuttis/cppmodules`. I hope Microsoft will fix their flaws so that this workaround is no longer necessary.

- **gcc/g++:**
  gcc does not require any special file extension or command-line option at all. Therefore, by using special file extensions, you only have to specify that the files contain C++ code using the command-line option `-xc++`:
  - Compile an interface file `file.cppm` as follows:
    ```
    g++ -xc++ -c file.cppm
    ```
  - Compile an internal partition file `file.cppp` as follows:
    ```
    g++ -xc++ -c file.cppp
    ```
- **Clang:**
  Clang currently supports only interface files. As the proposed extension **`.cppm`** is required for them anyway, using it should work.
  However, you cannot use internal partition files.

16.3.2 Dealing with Header Files

While in theory modules could replace all traditional header files with all their flaws, in practice this will never happen. There will be header files from code and libraries developed for C++ (and C) that do not require the use of modules. This will be the case especially because the use of a precompiler makes compiling and linking C++ programs more complex. Therefore, modules should be able to deal with traditional header files.

The basic approach for using traditional header files is to use the *global module fragment*.

- Start your module with `module;`
- Then, place all necessary preprocessor commands before you come to the module declaration

In that case:

- Everything not used from the header file included will be discarded.
- Everything used will get module linkage, which means that it is visible only throughout the module unit but neither in other module units nor outside the module.
- A `#define` before the `#include` is honored.

For example:

```
module;

#include <string>
#define NDEBUG
#include <cassert>

export module ModTest;
...
void foo(std::string s) {
  assert(s.empty());    // valid but not checked
  ...
}
```

With this global module fragment, the preprocessor symbol `NDEBUG` and the macro `assert()` from `<cassert>` are defined within this module unit. However, due to `NDEBUG`, any runtime check with `assert()` is disabled.

Both `NDEBUG` and `assert()` are not visible in other units of this module or imported modules.

After the declaration of a module, `#include` is no longer supported. Other preprocessor commands such as `#define` and `#ifdef` can be used.

`import` for Header Files

The future goal is to have the whole C++ standard library available as modules. However, for the standard C++ header files, it is already possible to use `import`, which then can be used within a module. For example:

```
export module ModTest;

import <chrono>;
```

That command is a shortcut for declaring and importing a module with everything from the corresponding header file exported. Even macros are visible within this module with this `import` (with all other imports, they are not).

However, constants that are defined before the import with `#define` are not passed to the imported header file. That way, we can guarantee that the content of the imported header file is always the same so that the header file can be precompiled.

Note that this feature is only guaranteed to work on standard C++ headers. It also does *not* apply to a standard C header adopted by C++:

```
export module ModTest;

import <chrono>;     // OK
import <cassert>;    // ERROR (or at least not portable)
```

Platforms are also allowed to support this feature for other header files; however, code using this feature is not portable.

Standard Modules

C++20 just introduces the technology. It does not introduce any standard modules (one reason is that the standards committee wants to reorganize the symbols over different modules to clean up some of the historic mess with header files).

It looks like with C++23, there will be two standard modules (see `http://wg21.link/p2465`):

- **`std`** will provide everything in the namespace `std` from C++ headers, including those that wrap C (e.g., `std::sort()`, `std::ranges::sort()`, `std::fopen()`, and `::operator new`).

  No macros and no feature test macros are provided. For them, you have to include `<cassert>` or `<version>` yourself.
- **`std.compat`** will provide everything from the module `std` plus the counterparts of C symbols from C headers (e.g., `::fopen()`).

Note that `std` and every module name starting with `std` is reserved by the C++ standard for its standard modules.

16.4 Modules in Detail

This section describes a few additional details for using modules.

16.4.1 Private Module Fragments

If you declare a module in a primary interface, you might sometimes need a *private module fragment*. This allows programmers to have declarations and definitions within the primary interface that are neither visible nor reachable by any other module or translation unit. One way to use a private module fragment is to disable exporting the definition of a class or function although its declaration is exported.

Consider, for example, the following primary interface:

```
export module MyMod;

export class C;                     // class C is exported
export void print(const C& c);     // print() is exported

class C {                           // provides details of the exported class
 private:
  int value;
 public:
  void print() const;
};

void print(const C& c) {            // provides details of the exported function
  c.print();
}
```

Here, we first forward declare the class C and function print() with export:

```
export module MyMod;

export class C;                     // class C is exported
export void print(const C& c);     // print() is exported
```

export may be specified only once at the point where a name is introduced in its namespace. The details are also exported later. Therefore, any translation unit can import this module and use objects of type C:

```
import MyMod;
...

C c;          // OK, definition of class C was exported
print(c);     // OK (compiler can replace the function call with its body)
```

However, if you want to encapsulate the definition within the module, so that importing code sees only the declaration and you still want to have the definition in the primary interface, you have to place the definitions in the *private module fragment*:

```
export module MyMod;

export class C;                    // declaration is exported
export void print(const C& c);     // declaration is exported

module :private;    // following symbols are not even implicitly exported

class C {                          // complete class not exported
 private:
  int value;
 public:
  void print() const;
};

void print(const C& c) {           // definition not exported
  c.print();
}
```

The *private module fragment* is declared with

```
module :private;
```

It can only occur in primary interfaces and can occur only once. With its declaration, the rest of the file is no longer implicitly exported (not even implicitly). Using `export` afterwards to export anything is an error.

By moving the definition into the *private module fragment*, importing code can no longer use any definition from there. It can only use the forward declaration of the class `C` (class `C` is an *incomplete type*) and `print()`.

For example, you cannot create objects of type `C`:

```
import MyMod;
...

C c;        // ERROR (C only declared, not defined)
print(c);   // OK (compiler can replace the function call with its body)
```

However, the declarations are good enough to use references and pointers of type `C`:

```
import MyMod;
...

void foo(const C& c) {  // OK
  print(c);             // OK
}
```

16.4.2 Module Declaration and Export in Detail

Module units have to start with one of the following (after initial comments and whitespace):

- `module;`
- `export module` *name*`;`
- `module` *name*`;`
- `module` *name*`:`*partname*`;`
- `export module` *name*`:`*partname*`;`

If a module unit starts with `module;` to introduce a global module fragment, exactly one of the other module declarations must follow the preprocessor commands in the global module fragment.

Within the module, you can `export` all kinds of symbols that have a name:

- You can export namespaces that have a name, which exports all symbols defined within the namespace declaration. For example:

```
export namespace MyMod {
  ...     // exported symbols of namespace MyMod
}

namespace MyMod {
  ...     // symbols of namespace MyMod not exported
}

export namespace MyMod {
  ...     // more exported symbols of namespace MyMod
}
```

- You can export types, which exports them with all members (if there are any). For example:

```
export class MyClass;
export struct MyStruct;
export union MyUnion;
export enum class MyEnum;
export using MyString = std::string;
```

  You do not have to export class members or enumeration values. Class members and values of enumeration types are automatically exported if the type is exported.

- You can export objects. For example:

```
export std::string progname;

namespace MyStream {
  export using std::cout;   // export std::cout as MyStream::cout
}

export auto myLambda = [] {};
```

- You can export functions. For example:

```
export friend std::ostream& operator<< (std::ostream&, const MyType&);
```

To declare an entity that shall be exported, use `export` for its first declaration. You can later specify again that it is exported, but declaring an entity without `export` and declaring/defining it later with `export` is not allowed.[2]

`export` is not allowed in unnamed namespaces, for `static` objects, and within a *private module fragment*.

No `inline` is necessary for any `export`. Formally, a definition within a module always exists only once. This applies even if the object is also re-exported by another module.

16.4.3 Umbrella Modules

Modules can export everything they import. For imported interface partitions, `export` is even required.

To export imported symbols, you can usually use `using`:

```
export module MyMod;                // declare module MyMod

// export all symbols from OtherModule as a whole:
export import OtherModule;

// import LogModule to export parts of it:
import LogModule

// export Logger in the namespace LogModule as ::Logger:
export using LogModule::Logger;

// export Logger in the namespace LogModule as LogModule::Logger:
export namespace LogModule {
  using LogModule::Logger;
}

// export global symbol globalLogger:
export using ::globalLogger;

// export global symbol log (e.g., function log()):
export using ::log;
```

[2] However, I have seen compilers accepting that.

16.4.4 Module Import in Detail

With **import**, any C++ source code file can import a module to use the functions, types, and objects exported there.

import is not an ordinary keyword; it is a contextual keyword. This means that you can still name other components with the identifier import, although this is not recommended. It was not introduced as a keyword because that might have broken too much existing code.

Translation or module units that use import have to be compiled after the module has been precompiled. Otherwise, you might get an error that the module is not defined or, even worse, you might compile against an old version of a module. As a consequence, circular imports are not possible.

16.4.5 Reachable versus Visible Symbols in Detail

Let us look at a few more details and an example of the visibility and reachability of exported and imported symbols.

When importing a module, you also indirectly import all types that are used by the exported API. If these types are not exported explicitly, you can use the types including all of their member functions, but no free-standing functions.

The following module exports getPerson() as a visible symbol and Person as a reachable class:

modules/person1.cppm

```
module;
#include <iostream>
#include <string>

export module ModPerson;      // THE module interface

class Person {                // note: not exported
   std::string name;
 public:
   Person(std::string n)
    : name{std::move(n)} {
   }
   std::string getName() const {
     return name;
   }
};

std::ostream& operator<< (std::ostream& strm, const Person& p)
{
  return strm << p.getName();
}

export Person getPerson(std::string s) {
  return Person{s};
}
```

Importing this module has the following consequences:

```
#include <iostream>
import ModPerson;    // import module ModPerson
...

Person p1{"Cal"};                  // ERROR: Person not visible
Person p2 = getPerson("Kim");      // ERROR: Person not visible
auto p3 = getPerson("Tana");       // OK
std::string s1 = p3.getName();     // ERROR (unless <iostream> includes <string>)
auto s2 = p3.getName();            // OK
std::cout << p3 << '\n';           // ERROR: free-standing operator<< not exported
std::cout << s2 << '\n';           // OK
```

If this code also includes the header file for strings, the declaration of `s1` compiles:

```
#include <iostream>
#include <string>
import ModPerson;    // import module ModPerson
...

Person p1{"Cal"};                  // ERROR: Person not visible
Person p2 = getPerson("Kim");      // ERROR: Person not visible
auto p3 = getPerson("Tana");       // OK
std::string s1 = p3.getName();     // OK
auto s2 = p3.getName();            // OK
std::cout << p3 << '\n';           // ERROR: free-standing operator<< not exported
std::cout << s2 << '\n';           // OK
```

If you declare `operator<<` within the class `Person` as a hidden `friend` (something you should always do):

```
export module ModPerson;

class Person {
 ...
 friend std::ostream& operator<< (std::ostream& strm, const Person& p) {
   return strm << p.getName();
 }
};
```

the member operator becomes reachable:

```
auto p3 = getPerson("Tana");       // OK
std::cout << p3 << '\n';           // OK (operator<< is reachable now)
```

Note again that by using *private module fragments*, you can restrict the reachability of indirectly exported symbols.

Non-Exported Symbols Cannot Conflict

The behavior that indirectly exported symbols are not visible but reachable allows programmers to use different modules that use the same symbol names in their exported interface. Consider, for example, you have a module defining a class `Person` as follows:

```
export module ModPerson1;

class Person {
 ...
 public:
  std::string getName() const {
    return name;
  }
};

export Person getPerson1(std::string s) {
  return Person{s};
}
```

and another module also defining a different class `Person`:

```
export module ModPerson2;

class Person {
 ...
 public:
  std::string getName() const {
    return name;
  }
};

export Person getPerson2(std::string s) {
  return Person{s};
}
```

In that case, a program can import both modules without any conflict because the only visible symbols are `getPerson1()` from the first module and `getPerson2()` from the second module. The following code works fine:

```
auto p1 = getPerson1("Tana");
auto s1 = p1.getName();

auto p2 = getPerson2("Tana");
auto s2 = p2.getName();
```

The types of `p1` and `p2` have the same name but are different:

```
std::same_as<decltype(p1), decltype(p2)>    // yields false
```

16.5 Afternotes

The idea of supporting modules in C++ is quite old. The first paper about this was published in 2004 by Daveed Vandevoorde in `http://wg21.link/n1736`.

Due to the complexity of the feature, a *Modules TS* (experimental technical specification) was established with `http://wg21.link/n4592` to work on details. Gabriel Dos Reis was the main driver for the content of this TS.

The wording to finally merge modules into the C++20 standard was formulated by Richard Smith in `http://wg21.link/p1103r3`. After that, a couple of fixes and clarifications were applied by various authors in various papers.

This page is intentionally left blank

Chapter 17

Lambda Extensions

This chapter presents the supplementary features that C++20 introduces for lambdas.

17.1 Generic Lambdas with Template Parameters

C++20 introduces an extension to enable the use of template parameters for generic lambdas. You can specify these template parameters between the capture clause and the call parameters (if there are any):

```
auto foo = []<typename T>(const T& param) {  // OK since C++20
               T tmp{};   // declare object with type of the template parameter
               ...
           };
```

Template parameters for lambdas provide the benefit of having a name for the type or a part of it when declaring generic parameters. For example:

```
[]<typename T>(T* ptr) {  // OK since C++20
  ...     // can use T as type of the value that ptr points to
};
```

Or:

```
[]<typename T, int N>(T (&arr)[N]) {
  ...     // can use T as element type and N as size of the passed array
};
```

If you are wondering why you would not use a function template in these cases, remember that lambdas provide some benefits that functions cannot provide:

- They can be defined inside functions.
- They can capture runtime values to specify their functional behavior at runtime.
- You can pass them as arguments without the need to specify the parameter types.

17.1.1 Using Template Parameters for Generic Lambdas in Practice

Explicit template parameters can be useful for specializing (or partially restricting the type of parameters of) generic lambdas. Consider the following example:

```
[]<typename T>(const std::vector<T>& vec) {  // can only pass vectors
  ...
};
```

This lambda accepts only vectors as arguments. When using `auto`, it would not be easy to restrict the argument to vectors because C++ does not (yet) support something like `std::vector<auto>`. However, in this case, you could also use type constraints to constrain the type of the parameter (such as to require random access or even a specific type).

Explicit template parameters also help to avoid the need for `decltype`. For example, for perfect forwarding of a generic parameter pack in a lambda, you can write:

```
[]<typename... Types>(Types&&... args) {
  foo(std::forward<Types>(args)...);
};
```

instead of:

```
[] (auto&&... args) {
  foo(std::forward<decltype(args)>(args)...);
};
```

A similar example would be code to provide special behavior of a certain type when visiting a `std::variant<>` (introduced in C++17):

```
std::variant<int, std::string> var;
...
// call generic lambda with type-specific behavior:
std::visit([](const auto& val) {
             if constexpr(std::is_same_v<decltype(val), const std::string&>) {
               ...     // string-specific processing
             }
             std::cout << val << '\n';
           },
           var);
```

We have to use `decltype()` to get the type of the parameter and compare that type as a `const&` (or remove `const` and the reference). Since C++20, you can just write the following:

```
std::visit([]<typename T>(const T& val) {  // since C++20
             if constexpr(std::is_same_v<T, std::string>) {
               ...     // string-specific processing
             }
             std::cout << "value: " << val << '\n';
           },
           var);
```

You can also declare template parameters for `consteval` lambdas to force that their execution at compile time.

17.1.2 Explicit Specification of Lambda Template Parameters

Lambdas provide a convenient way to define function objects (functors). For generic lambdas, their function call operator (`operator()`) is a template. With the syntax to specify a name for the parameter instead of using `auto`, you have a name for the template parameter in the generated function call operator.

For example, if you define the following lambda:

```
auto primeNumbers = [] <int Num> () {
                      std::array<int, Num> primes{};
                      ...   // compute and assign first Num prime numbers
                      return primes;
                    };
```

the compiler defines a corresponding *closure type*:

```
class NameChosenByCompiler {
 public:
  ...
  template<int Num>
  auto operator() () const {
    std::array<int, Num> primes{};
    ...   // compute and assign first Num prime numbers
    return primes;
  }
};
```

and creates an object of this class (using the default constructor if no values are captured):

```
auto primeNumbers = NameChosenByCompiler{};
```

To specify the template parameter explicitly, you have to pass it to `operator()` when you use the lambda as a function:

```
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers.operator()<20>();
```

There is no way to avoid specifying `operator()` when specifying its template parameters except by using an indirect call.

You can try to make the template parameter deducible by making it a compile-time value; however, the resulting syntax is not much better:[1]

```
auto primeNumbers = [] <int Num> (std::integral_constant<int, Num>) {
                     std::array<int, Num> primes{};
                     ...   // compute and assign first Num prime numbers
                     return primes;
                    };

// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers(std::integral_constant<int,20>{});
```

[1] Thanks to Arthur O'Dwyer and Jonathan Wakely for pointing this out.

Alternatively, you might think about using a *variable template*, a technique introduced in C++14. With this, you can make the variable `primeNumbers` generic instead of making the lambda generic:

```
template<int Num>
auto primeNumbers = [] () {
                      std::array<int, Num> primes{};
                      ...   // compute and assign first Num prime numbers
                      return primes;
                    };
...
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers<20>();
```

However, in that case, you cannot define the lambda inside a function scope. Generic lambdas allow you to define generic functionality locally inside a scope.

17.2 Calling the Default Constructor of Lambdas

Lambdas provide a simple way to define function objects. If you define

```
auto cmp = [] (const auto& x, const auto& y) {
             return x > y;
           };
```

this is equivalent to defining a class (the *closure type*) and creating an object of this class:

```
class NameChosenByCompiler {
 public:
  template<typename T1, T2>
  auto operator() (const T1& x, const T2& y) const {
    return x > y;
  }
};

auto cmp = NameChosenByCompiler{};
```

The generated closure type has `operator()` defined, which means that you can use the lambda object `cmp` as a function:

```
cmp(val1, val);    // yields the result of 42 > obj2
```

However, before C++20, the generated closure type had no callable default constructor and assignment operator. Objects of the generated class could only be initially created by the compiler. Only copying was possible:

```
auto cmp1 = [] (const auto& x, const auto& y) {
              return x > y;
            };

auto cmp2 = cmp1;       // OK, copy constructor supported since C++11
decltype(cmp1) cmp3;    // ERROR until C++20: no default constructor provided
cmp1 = cmp2;            // ERROR until C++20: no assignment operator provided
```

As a consequence, you could not easily pass a lambda as a sorting criterion or hash function to a container, where the ***type*** of the helper function was required. Consider a class `Customer` with the following interface:

```
class Customer
{
 public:
  ...
  std::string getName() const;
};
```

To use the name returned by `getName()` as the ordering criterion or value for the hash function, you had to pass both the type and the lambda as a template and call parameter:

```
// create balanced binary tree with user-defined ordering criterion:
auto lessName = [] (const Customer& c1, const Customer& c2) {
                  return c1.getName() < c2.getName();
                };
std::set<Customer, decltype(lessName)> coll1{lessName};

// create hash table with user-defined hash function:
auto hashName = [] (const Customer& c) {
                  return std::hash<std::string>{}(c.getName());
                };
std::unordered_set<Customer, decltype(hashName)> coll2{0, hashName};
```

The containers get the lambda when they are initialized so that they can use an internal copy of the lambda (for the unordered containers, you have to pass a minimum bucket size before). To compile, the type of the containers needs the type of the lambda.

Since C++20, lambdas with no captures have a default constructor and an assignment operator:

```
auto cmp1 = [] (const auto& x, const auto& y) {
              return x > y;
            };

auto cmp2 = cmp1;       // OK, copy constructor supported
decltype(cmp1) cmp3;    // OK since C++20
cmp1 = cmp2;            // OK since C++20
```

For this reason, it is now enough to pass the type of the lambda for the ordering criterion or hash function respectively:

```
// create balanced binary tree with user-defined ordering criterion:
auto lessName = [] (const Customer& c1, const Customer& c2) {
                  return c1.getName() < c2.getName();
                };
std::set<Customer, decltype(lessName)> coll1;           // OK since C++20

// create hash table with user-defined hash function:
```

```
auto hashName = [] (const Customer& c) {
                  return std::hash<std::string>{}(c.getName());
                };
std::unordered_set<Customer, decltype(hashName)> coll2;   // OK since C++20
```

This works because the argument for the ordering criterion or hash function has a default value, which is a default-constructed object of the type of the ordering criterion or hash function. And because lambdas without captures have a default constructor since C++20, the initialization of the ordering criterion with a default-constructed object of the type of the lambda now compiles.

You can even define the lambda inside the declaration of the container and use `decltype` to pass its type. For example, you can declare an associative container with an ordering criterion defined in the declaration as follows:

```
// create balanced binary tree with user-defined ordering criterion:
std::set<Customer,
         decltype([] (const Customer& c1, const Customer& c2) {
                    return c1.getName() < c2.getName();
                  })> coll3;                            // OK since C++20
```

In the same way, you can declare an unordered container with a hash function defined in the declaration as follows:

```
// create hash table with user-defined hash function:
std::unordered_set<Customer,
                   decltype([] (const Customer& c) {
                              return std::hash<std::string>{}(c.getName());
                            })> coll;                  // OK since C++20
```

See *lang/lambdahash.cpp* for a complete example.

17.3 Lambdas as Non-Type Template Parameters

Since C++20, lambdas can be used as non-type template parameters (NTTPs):

```
template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

auto defaultTax = [] {  // OK
  return 0.19;
};

std::cout << addTax<defaultTax>(100) << '\n';
```

This feature is a side effect of the new support for using literal types with public members only as non-type template parameter types. See the chapter about non-type template parameter extensions for a detailed discussion and complete example.

17.4 consteval Lambdas

By using the new consteval keyword with lambdas, you can now require that lambdas become *immediate functions* so that “function calls” of them have to be evaluated at compile time. For example:

```
auto hashed = [] (const char* str) consteval {
                  ...
                };

auto hashWine = hashed("wine");   // hash() called at compile time
```

Due to the use of consteval in the definition of the lambda, any call has to happen at compile time with values known at compile time. Passing a runtime value is an error:

```
const char* s = "beer";
auto hashBeer = hashed(s);              // ERROR

constexpr const char* cs = "water";
auto hashWater = hashed(cs);            // OK
```

Note that hashed itself does not have to be constexpr. It is a runtime object of the lambda for which the “function call” is performed at compile time.

The discussion of consteval for lambdas in the section about the new consteval keyword provides more details of this example.

You can also use the new template syntax for generic lambdas with consteval. This enables programmers to define the initialization of a compile-time function inside another function. For example:

```
// local compile-time computation of Num prime numbers:
auto primeNumbers = [] <int Num> () consteval {
                      std::array<int, Num> primes;
                      int idx = 0;
                      for (int val = 1; idx < Num; ++val) {
                        if (isPrime(val)) {
                          primes[idx++] = val;
                        }
                      }
                      return primes;
                    };
```

See *lang/lambdaconsteval.cpp* for a complete program using this lambda.

Note that in this case, the template parameter is not deduced. Therefore, the syntax to explicitly specify the template parameter becomes a bit ugly:

```
auto primes = primeNumbers.operator()<100>();
```

Note also that you always have to provide the parameter list before consteval (the same applies when specifying constexpr there). You cannot skip the parentheses even if there is no parameter declared.

17.5 Changes for Capturing

C++20 introduces several new extensions to the capturing of values and objects in lambdas.

17.5.1 Capturing `this` and `*this`

If a lambda is defined inside a member function, the question is how you can access data of the object for which the member function is called. Before C++20, we had the following rules:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); };             // ERROR: member name not captured
    auto l1 = [val, name=name] { bar(val, name); }; // OK, capture val and name by value

    auto l2 = [&] { bar(val, name); };               // OK (val and name by reference)
    auto l3 = [&, this] { bar(val, name); };         // OK (val and name by reference)
    auto l4 = [&, *this] { bar(val, name); };        // OK (val by reference, name by value)

    auto l5 = [=] { bar(val, name); };               // OK (val by value, name by reference)
    auto l6 = [=, this] { bar(val, name); };         // ERROR before C++20
    auto l7 = [=, *this] { bar(val, name); };        // OK (val and name by value)
    ...
  }
};
```

Since C++20, the following rules apply:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); };        // ERROR: member name not captured
    auto l1 = [val, name=name] { bar(val, name); }; // OK, capture val and name by value

    auto l2 = [&] { bar(val, name); };          // deprecated (val and name by ref.)
    auto l3 = [&, this] { bar(val, name); };    // OK (val and name by reference)
    auto l4 = [&, *this] { bar(val, name); };   // OK (val by reference, name by value)

    auto l5 = [=] { bar(val, name); };          // deprecated (val by value, name by ref.)
```

```
      auto l6 = [=, this] { bar(val, name); };    // OK (val by value, name by reference)
      auto l7 = [=, *this] { bar(val, name); };   // OK (val and name by value)
      ...
    }
  };
```

Thus, since C++20, we have the following changes:

- `[=, this]` is now allowed as a lambda capture (some compilers did allow it before, although it was formally invalid).
- The implicit capture of `*this` is deprecated.

17.5.2 Capturing Structured Bindings

Since C++20, it the capture of structured bindings (introduced with C++17) is allowed:

```
std::map<int, std::string> mymap;
...

for (const auto& [key,val] : mymap) {
   auto l = [key, val] {   // OK since C++20
                ...
            };
   ...
}
```

Some compilers did allow the capture of structured bindings before, although it was formally invalid.

17.5.3 Capturing Parameter Packs of Variadic Templates

If you have a variadic template, you could capture parameter packs as follows:

```
template<typename... Args>
void foo(Args... args)
{
   auto l1 = [&] {
               bar(args...);   // OK
             };
   auto l2 = [args...] {       // or [=]
               bar(args...);   // OK
             };
   ...
}
```

However, if you wanted to return the lambda that was created for later use, there was a problem:

- Using `[&]`, you would return a lambda that refers to a destroyed parameter pack.
- Using `[args...]` or `[=]`, you would copy the passed parameter pack.

You can usually use *init-captures* to use move semantics when capturing objects:

```
template<typename T>
void foo(T arg)
{
   auto l3 = [arg = std::move(arg)] {  // OK since C++14
                bar(arg);        // OK
              };
   ...
}
```

However, there was no syntax provided to use *init-captures* with parameter packs.

C++20 introduces a corresponding syntax:

```
template<typename... Args>
void foo(Args... args)
{
   auto l4 = [...args = std::move(args)] {  // OK since C++20
                bar(args...);   // OK
              };
   ...
}
```

You can also init-capture parameter packs by reference. For example, we can change to a parameter name for the lambda as follows:

```
template<typename... Args>
void foo(Args... args)
{
   auto l4 = [&...fooArgs = args] {          // OK since C++20
                bar(fooArgs...);  // OK
              };
   ...
}
```

Example of Capturing Parameter Packs of Variadic Templates

A generic function that creates and returns a lambda that captures a variadic number of parameters by value now looks as follows:

```
template<typename Callable, typename... Args>
auto createToCall(Callable op, Args... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
         };
}
```

Using the new syntax for abbreviated function templates, the code looks as follows:

```
auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
         };
}
```

Here is a complete example:

lang/capturepack.cpp

```
#include <iostream>
#include <string_view>

auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
           return op(args...);
         };
}

void printWithGAndNoG(std::string_view s)
{
  std::cout << s << "g " << s << '\n';
}

int main()
{
  auto printHero = createToCall(printWithGAndNoG, "Zhan");
  ...

  printHero();
}
```

Run it to print a hero!

17.5.4 Lambdas as Coroutines

Lambdas can also be coroutines, which were introduced with C++20. However, note that in this case, the lambdas should not capture anything, because a coroutine may easily be used longer than the lambda object, which is created locally, exists.

17.6 Afternotes

The template syntax for generic lambdas was first proposed by Louis Dionne in `http://wg21.link/p0428r0`. The finally accepted wording was formulated by Louis Dionne in `http://wg21.link/p0428r2`.

The template syntax for generic lambdas was first proposed by Louis Dionne in `http://wg21.link/p0624r0`. The finally accepted wording was formulated by Louis Dionne in `http://wg21.link/p0624r2`.

Allowing lambdas as non-type template parameters was introduced as finally formulated by Jeff Snyder and Louis Dionne in `http://wg21.link/p0732r2`.

Support for `consteval` lambdas was introduced as finally formulated by Richard Smith, Andrew Sutton, and Daveed Vandevoorde in `http://wg21.link/p1073r3`.

The changes to the rules for capturing `this` and `*this` were first proposed by Thomas Köppe in `http://wg21.link/p0409r0` and `http://wg21.link/p0806r0`. The finally accepted wording was formulated by Thomas Köppe in `http://wg21.link/p0409r2` and `http://wg21.link/p0806r2`.

Capturing structured bindings was introduced as finally formulated by Nicolas Lesser in `http://wg21.link/p1091r3`.

Init-capturing parameter packs was accepted as finally formulated by Barry Revzin in `http://wg21.link/p0780r2` and `http://wg21.link/p2095r0`.

Chapter 18

Compile-Time Computing

This chapter presents several extensions of C++ to support compile-time computing.

The chapter covers the two new keywords `constinit` and `consteval`, and the extensions that allow programmers to use heap memory, vectors, and strings at compile time.

18.1 Keyword `constinit`

One new keyword C++20 introduces is **`constinit`**. It can be used to force and ensure that a *mutable* static or global variable is initialized at compile time. Roughly speaking, the effect is described as:[1]

```
constinit = constexpr - const
```

Yes, a `constinit` variable is ***not* `const`** (it would have better to name the keyword `compiletimeinit`). The name comes from the fact that these initializations usually happen when compile-time constants are initialized.

You can use `constinit` whenever you declare a static or global variable. For example:

```
// outside any function:
constinit auto i = 42;

int getNextClassId() {
  static constinit int maxId = 0;
  return ++maxId;
}

class MyType {
  static constinit long max = sizeof(int) * 1000;
  ...
};
```

[1] Thanks to Jonathan Müller for pointing this out.

```
constexpr std::array<int, 5> getColl() {
  return {1, 2, 3, 4, 5};
}
constinit auto globalColl = getColl();
```

As written, you can still modify the declared values. The following code using the declarations above for the first time:

```
std::cout << i << " " << coll[0] << '\n';   // prints 42 1
i *= 2;
coll = {};
std::cout << i << " " << coll[0] << '\n';   // prints 84 0
```

has the following output:

```
42 1
84 0
```

The effect of using `constinit` is that the initialization compiles only if the initial value is a constant value known at compile time. This means that in contrast to a declaration such as:

```
auto x = f();               // f() might be a runtime function
```

the corresponding declaration with `constinit` requires a compile-time initialization, meaning that it must be possible to call `f()` at compile time (which means `f()` must be `constexpr` or `consteval`).

```
constinit auto x = f();     // f() must be a compile-time function
```

If you initialize an object with `constinit`, it must be possible to use the constructor at compile time:

```
constinit std::pair p{42, "ok"};    // OK
constinit std::list l;              // ERROR: default constructor not constexpr
```

The reasons for using `constinit` are as follows:

- You can require initialization of mutable global/static objects at compile time. That way, you can avoid the initialization using runtime. In particular, this can improve performance when using `thread_local` variables.
- You can ensure that a global/static object is always initialized when it is used. In fact, `constinit` can be used to fix the *static initialization order fiasco*, which can occur when an initial value of a static/global object depends on another static/global object.

Note that using `constinit` never changes the functional behavior of a program (unless we have the static initialization order fiasco). It can only lead to the consequence that code no longer compiles.

18.1.1 Using constinit in Practice

There are couple of thing to respect when using constinit.[2]

First, you cannot initialize a constinit value with another constinit value:

```
constinit auto x = f();    // f() must be a compile-time function
constinit auto y = x;      // ERROR: x is not a constant initializer
```

The reason is that the initial value must be a constant value known at compile time, but constinit values are not constant. Only the following compiles:

```
constexpr auto x = f();    // f() must be a compile-time function
constinit auto y = x;      // OK
```

When initializing objects, a compile-time constructor is required. However, a compile-time ***destructor*** is not required. For this reason, you can use constinit for smart pointers:

```
constinit std::unique_ptr<int> up;   // OK
constinit std::shared_ptr<int> sp;   // OK
```

constinit does not imply inline (this is different from constexpr). For example:

```
class Type {
  constinit static int val1 = 42;          // ERROR
  inline static constinit int val2 = 42;   // OK
  ...
};
```

You can use constinit together with extern:

```
// header:
extern constinit int max;

// translation unit:
constinit int max = 42;
```

For the same effect, you can also skip constinit in the declaration. However, skipping constinit in the definition would no longer force compile-time initialization.

You can use constinit together with static and thread_local:

```
static thread_local constinit int numCalls = 0;
```

Any order of constinit, static, and thread_local is fine.

Note that for thread_local variables, using constinit might create a performance improvement, because it might avoid that the generated code needs an internal guard to signal whether the variable is already initialized:

```
extern thread_local int x1 = 0;
extern thread_local constinit int x2 = 0;   // better (might avoid an internal guard)
```

Using constinit to declare references is possible but makes no sense because the reference refers to a constant object. You should use constexpr instead.

[2] Thanks to http://cppreference.com for pointing out some of the issues mentioned here.

18.1.2 How `constinit` Solves the Static Initialization Order Fiasco

In C++, there is a problem called *static initialization order fiasco*, which `constinit` can solve. The problem is that the order of static and global initializations in different translation units is not defined. For that reason, the following code can be a problem:

- Assume we have a type with a constructor to initialize the objects and introduce an extern global object of this type:

  comptime/truth.hpp

  ```
  #ifndef TRUTH_HPP
  #define TRUTH_HPP

  struct Truth {
    int value;
    Truth() : value{42} {   // ensure all objects are initialized with 42
    }
  };

  extern Truth theTruth;     // declare global object

  #endif // TRUTH_HPP
  ```

- We initialize the object in its own translation unit:

  comptime/truth.cpp

  ```
  #include "truth.hpp"

  Truth theTruth;            // define global object (should have value 42)
  ```

- And then in another translation unit, we initialize another global/static object with `theTruth`:

  comptime/fiasco.cpp

  ```
  #include "truth.hpp"
  #include <iostream>

  int val = theTruth.value;      // may be initialized before theTruth is initialized

  int main()
  {
    std::cout << val << '\n';    // OOPS: may be 0 or 42
    ++val;
    std::cout << val << '\n';    // OOPS: may be 1 or 43
  }
  ```

There is a good chance that val is initialized with theTruth ***before*** theTruth itself was initialized. As a result, the program might have the following output:[3]

```
0
1
```

When using constinit to declare val, that problem cannot occur. constinit ensures that an object is *always* initialized before it is used because the initialization happens at compile time. If the guarantee cannot be given, the code does not compile. Note that an initialization would also be guaranteed if val was declared with constexpr; however, in that case, you would not be able to modify the value anymore.

In our example, just using constinit would first result in a compile-time error (signaling that initialization cannot be guaranteed at compile time):

```
// truth.hpp:
struct Truth {
  int value;
  Truth() : value{42} {
  }
};
extern Truth theTruth;

// main translation unit:
constinit int val = theTruth.value ;   // ERROR: no constant initializer
```

That error message signals now at compile time that val cannot be initialized at compile time. To make the initialization valid, you have to modify the declaration of class Truth and theTruth so that theTruth can be used at compile time:

comptime/truthc.hpp

```
#ifndef TRUTH_HPP
#define TRUTH_HPP

struct Truth {
  int value;
  constexpr Truth() : value{42} {   // enable compile-time initialization
  }
};

constexpr Truth theTruth;             // force compile-time initialization

#endif // TRUTH_HPP
```

[3] For example, when using the gcc compiler, you get this effect if you pass truth.o before fiasco.o to the linker.

Now, the program compiles and `val` is guaranteed to be initialized with the initialized value of `theTruth`:

comptime/constinit.cpp

```
#include "truthc.hpp"
#include <iostream>

constinit int val = theTruth.value ;   // initialized after theTruth is initialized

int main()
{
  std::cout << val << '\n';     // guaranteed to be 42
  ++val;
  std::cout << val << '\n';     // guaranteed to be 43
}
```

Therefore, the output of the program is now guaranteed to be:

```
42
43
```

There are other ways to solve the static initialization order fiasco (using a static function to get the value or using `inline`). Nevertheless, you might think carefully about following a programming style that always declares global and static variables with `constinit`, provided the initialization does not need any runtime value/feature. Using it in a function like this at least does not hurt:

```
long nextId()
{
  constinit static long id = 0;
  return ++id;
}
```

18.2 Keyword consteval

Since C++11, C++ has the keyword `constexpr` to support the evaluation of functions at compile time. Provided all aspects of the functions are known at compile time, you can also use the results in compile-time contexts. However, `constexpr` functions also serve as “normal” runtime functions.

C++20 introduces a similar keyword **`consteval`**, which mandates compile-time computing. In contrast to functions marked with `constexpr`, functions marked with `consteval` cannot be called at runtime; instead, they are ***required*** to be called at compile time. If this is not possible, the program is ill-formed. Because these functions are called immediately, when the compiler sees calls of them, these functions are also called ***immediate functions***.

18.2.1 A First consteval Example

Consider the following example:

comptime/consteval1.cpp

```
#include <iostream>
#include <array>

constexpr
bool isPrime(int value)
{
  for (int i = 2; i <= value/2; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return value > 1;   // 0 and 1 are not prime numbers
}

template<int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  int idx = 0;
  for (int val = 1; idx < Num; ++val) {
    if (isPrime(val)) {
      primes[idx++] = val;
    }
  }
  return primes;
}

int main()
{
  // init with prime numbers:
  auto primes = primeNumbers<100>();

  for (auto v : primes) {
    std::cout << v << '\n';
  }
}
```

Here, we use `consteval` to define the function `primeNumbers<`*N*`>()`, which returns an array of the first *N* prime numbers at compile time:

```
template<int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
```

To compute the prime numbers, `primeNumbers()` uses a helper function `isPrime()` that is declared with `constexpr` to be usable at both runtime and compile time (we could also declare it with `consteval`, but then it would not be usable at runtime).

The program then uses `primeNumbers<>()` to initialize an array of 100 prime numbers:

```
auto primes = primeNumbers<100>();
```

Because `primeNumbers<>()` is `consteval`, this initialization must happens at compile time. You would have the same effect if `primeNumbers<>()` is declared with `constexpr` and the function is called in a compile-time context (such as to initialize a `constexpr` or `constinit` array `prime`):

```
template<int Num>
constexpr
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
...

constinit static auto primes = primeNumbers<100>();
```

In both cases, you can see that the compile time becomes significantly slower when the number of prime numbers to compute for the initializations grows significantly.

However, note that compilers usually have a limit when evaluating constant expressions. The C++ standard only guarantees the evaluation of 1,048,576 expressions within a core constant expression.

`consteval` Lambdas

You can also declare lambdas to be `consteval` now. This requires that the lambda is evaluated at compile time.

Consider the following example:

```
int main(int argc, char* argv[])
{
  // compile-time function to compute hash value for string literals:
```

```
  // (for the algorithm, see http://www.cse.yorku.ca/~oz/hash.html)
  auto hashed = [] (const char* str) consteval {
                    std::size_t hash = 5381;
                    while (*str != '\0') {
                      hash = hash * 33 ^ *str++;
                    }
                    return hash;
                  };

  // OK (requires hashed() in compile-time context):
  enum class drinkHashes : long { beer = hashed("beer"), wine = hashed("wine"),
                                  water = hashed("water"), ... };

  // OK (hashed() guaranteed to be called at compile time):
  std::array arr{hashed("beer"), hashed("wine"), hashed("water")};

  if (argc > 1) {
    switch (hashed(argv[1])) {   // ERROR: argv is not known at compile time
      ...
    }
  }
}
```

Here we initialize `hashed` with a lambda that can be used only at compile time. Therefore, the use of the lambda has to happen in a compile-time context with compile-time values. The calls are valid only if they take string literals or a parameter of type `constexpr const char*`.

If you declare the lambda with `constexpr` the `switch` statement would become valid. (the declaration with `constexpr` is not necessary, because since C++17, all lambdas are implicitly `constexpr` if possible). Then, however, it is no longer guaranteed that the computation of the initial values for `arr` happens at compile time.

See the discussion on `consteval` lambdas in the chapter of all new lambda features for more details and another example.

18.2.2 constexpr versus consteval

With `constexpr` and `consteval`, we now have the following options for influencing when a function can be and is called:

- Neither `constexpr` nor `consteval`:
  These functions can be used only in runtime contexts. However, the compiler can still perform optimizations that evaluate them at compile time.
- `constexpr`:
  These functions can be used in compile-time and runtime contexts. Even in runtime contexts, the compiler can still perform optimizations that evaluate the functions at compile time. The compiler is also allowed to evaluate the functions in compile-time contexts at runtime.
- `consteval`:
  These functions can be used only at compile-time. However, the result can be used in a runtime context.

For example, consider the following three functions declared in a header file (therefore, `squareR()` is declared with `inline`):

comptime/consteval2.hpp

```cpp
// square() for runtime only:
inline int squareR(int x) {
  return x * x;
}

// square() for compile time and runtime:
constexpr int squareCR(int x) {
  return x * x;
}

// square() for compile time only:
consteval int squareC(int x) {
  return x * x;
}
```

We can use these functions as follows:

comptime/consteval2.cpp

```cpp
#include "consteval2.hpp"
#include <iostream>
#include <array>

int main()
{
  int i = 42;

  // using the square functions at runtime with runtime value:
  std::cout << squareR(i) << '\n';        // OK
  std::cout << squareCR(i) << '\n';       // OK
  //std::cout << squareC(i) << '';        // ERROR

  // using the square functions at runtime with compile-time value:
  std::cout << squareR(42) << '\n';       // OK
  std::cout << squareCR(42) << '\n';      // OK
  std::cout << squareC(42) << '\n';       // OK: square computed at compile time

  // using the square functions at compile time:
  //std::array<int, squareR(42)> arr1;    // ERROR
  std::array<int, squareCR(42)> arr2;     // OK: square computed at compile time
  std::array<int, squareC(42)> arr3;      // OK: square computed at compile time
  //std::array<int, squareC(i)> arr4;     // ERROR
}
```

The differences between constexpr and consteval are as follows:

- The consteval function is not allowed to process parameters that are not known at compile time:

```
  std::cout << squareCR(i) << '\n';      // OK
  std::cout << squareC(i) << '\n';       // ERROR
  std::array<int, squareC(i)> arr4;      // ERROR
```

- The consteval function must to perform its computing at compile time:

```
  std::cout << squareCR(42) << '\n';     // may be computed at compile time or runtime
  std::cout << squareC(42) << '\n';      // computed at compile time
```

This means that it makes sense to use consteval in two situations:

- You want to enforce compile-time computation.
- You want to disable a function to be used at runtime.

For example, whether a function square() or hashed() is constexpr or consteval makes no difference in compile-time contexts such as the following:

```
enum class Drink = { water = hashed("water"), wine = hashed("wine") };

switch (value) {
  case square(42):
    ...
}

if constexpr(hashed("wine") > hashed("water")) {
  ...
}
```

However, in runtime contexts, consteval might make a difference because compile-time computing is not required then:

```
std::array drinks = { hashed("water"), hashed("wine") };

std::cout << hashed("water");

if (hashed("wine") > hashed("water")) {
  ...
}
```

18.2.3 Using consteval in Practice

For consteval functions, a couple of restrictions apply when using them in practice.

Constraints for consteval

consteval functions share most of the other aspects of constexpr functions (note that these constraints were relaxed for constexpr functions with C++20):

- Parameters and the return type (if it is not void) have to be literal types.
- The body may contain only variables of literal types that are neither static nor thread_local.
- Using goto and labels is not allowed.
- The function may only be a constructor or destructor, the class has no virtual base class.
- consteval functions are implicitly inline.
- consteval functions cannot be used as coroutines.

Call Chains with consteval Functions

Functions marked with consteval can call other functions marked with constexpr or consteval:

```
constexpr int funcConstExpr(int i) {
  return i;
}

consteval int funcConstEval(int i) {
  return i;
}

consteval int foo(int i) {
  return funcConstExpr(i) + funcConstEval(i);  // OK
}
```

However, constexpr functions cannot call consteval functions for variables:

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  return funcConstEval(i);  // ERROR
}
```

The function foo() does not compile. The reason is that i is still not a compile-time value because it *could* be called at runtime. foo() could only call funcConstEval() with a compile-time variable:

```
constexpr int foo(int i) {
  return funcConstEval(42);  // OK
}
```

Note that even if(std::is_constant_evaluated()) does not help here.

Functions marked with consteval are also not permitted to call pure runtime functions (functions marked with neither constexpr nor consteval). However, this is only checked if the call is really performed. For a consteval function, it is not an error to contain statements that call runtime functions as long as they are not reached (this rule already applies to constexpr functions called at compile time).

Consider the following example:

```
void compileTimeError()
{
}

consteval int nextTwoDigitValue(int val)
{
  if (val < 0 || val >= 99) {
    compileTimeError();   // call something not valid to call at compile time
  }
  return ++val;
}
```

This compile-time function has the interesting effect that it can be used only for ***some*** arguments that have a value from zero to 98:

```
constexpr int i1 = nextTwoDigitValue(0);    // OK (initializes i1 with 1)
constexpr int i2 = nextTwoDigitValue(98);   // OK (initializes i2 with 99)
constexpr int i3 = nextTwoDigitValue(99);   // compile-time ERROR
constexpr int i4 = nextTwoDigitValue(-1);   // compile-time ERROR
```

Note that using static_assert() would not work here because it can only be called for values known at compile time and consteval does not make val a compile-time value inside the function:

```
consteval int nextTwoDigitValue(int val)
{
  static_assert(val >= 0 && val < 99);      // always ERROR: val is not a compile-time value
  ...
}
```

By using this trick, you can constrain compile-time functions to certain values. That way, you could signal an invalid format of a string parsed at compile time.

18.2.4 Compile-Time Value versus Compile-Time Context

You might assume that each and every value in a compile-time function is a compile-time value. However, that is not true. In compile-time functions, there is also a difference between static typing of the code and dynamic computing of values.

Consider the following example:

```
consteval void process()
{
  constexpr std::array a1{0, 8, 15};
  constexpr auto n1 = std::ranges::count(a1, 0);  // OK
```

```
    std::array<int, n1> a1b;                         // OK

    std::array a2{0, 8, 15};
    constexpr auto n2 = std::ranges::count(a2, 0);   // ERROR

    std::array a3{0, 8, 15};
    auto n3 = std::ranges::count(a3, 0);             // OK
    std::array<int, n3> a3b;                         // ERROR
  }
```

Although we are in a function that can be used only at compile time, `a2` is not a compile-time value. Therefore, it cannot be used where usually a compile-time value is required, such as to initialize the `constexpr` variable `n2`.

For the same reason, only `n1` can be used to declare the size of a `std::array`. Using `n3`, which is not `constexpr`, fails (even if `n3` were to be declared as `const`).

The same applies if `process()` is declared as a `constexpr` function. Whether it is called in a compile-time context does not matter.

18.3 Relaxed Constraints for `constexpr` Functions

Every C++ version from C++11 until C++20 relaxed the constraints for `constexpr` functions. This now also means that the `consteval` functions have these constraints.

Basically, the constraints for `constexpr` and `consteval` functions are now as follows:

- Parameters and the return type (if there is one) must be literal types.
- The body may define only variables of literal types. These variables may neither be `static` nor `thread_local`.
- Using `goto` and labels is not allowed.
- Constructors and destructors may be compile-time functions only if the class has no virtual base class.
- The functions cannot be used as coroutines.

The functions are implicitly `inline`.

18.4 `std::is_constant_evaluated()`

C++20 provides a new helper function that allows programmers to implement different code for compile-time and runtime computing: `std::`**`is_constant_evaluated()`**. It is defined in the header file `<type_traits>` (although it is not really a type function).

It allows code to switch from calling a helper function that can be called only at runtime to code that can be used at compile time. For example:

comptime/isconsteval.hpp

```
#include <type_traits>
#include <cstring>

constexpr int len(const char* s)
{
  if (std::is_constant_evaluated()) {
    int idx = 0;
    while (s[idx] != '\0') {        // compile-time friendly code
      ++idx;
    }
    return idx;
  }
  else {
    return std::strlen(s);          // function called at runtime
  }
}
```

In the function `len()`, we compute the length of a raw string or string literal. If called at runtime, we use the standard C function `strlen()`. However, to enable the function be used at compile time, we provide a different implementation if we are in a compile-time context.

Here is how the two branches can be called:

```
constexpr int l1 = len("hello");    // uses then branch
int l2 = len("hello");              // uses else branch (no required compile-time context)
```

The first call of `len()` happens in a compile-time context. In that case, `is_constant_evaluated()` yields `true` so that we use the *then* branch. The second call of `len()` happens in a runtime context, so that `is_constant_evaluated()` yields `false` and `strlen()` is called. The latter happens even if the compiler decides to evaluate the call at compile time. The important point is whether the calls is required to happen at compile time or not.

Here is a function that does the opposite: it converts an integral value to a string both at compile time and at runtime:

comptime/asstring.hpp

```
#include <string>
#include <format>

// convert an integral value to a std::string
// - can be called at compile time or runtime
constexpr std::string asString(long long value)
{
  if (std::is_constant_evaluated()) {
    // compile-time version:
    if (value == 0) {
      return "0";
    }
    if (value < 0) {
      return "-" + asString(-value);
    }
    std::string s = asString(value / 10) + std::string(1, value % 10 + '0');
    if (s.size() > 1 && s[0] == '0') {   // skip leading 0 if there is one
      s.erase(0, 1);
    }
    return s;
  }
  else {
    // runtime version:
    return std::format("{}", value);
  }
}
```

At runtime, we simply use `std::format()`. At compile time, we manually create a string of the optional negative sign and all digits (we use a recursive approach to bring them into the right order). An example of its use you can be found in the section about exporting compile-time strings to runtime strings.

18.4.1 `std::is_constant_evaluated()` in Detail

According to the C++20 standard, `std::is_constant_evaluated()` yields `true` when it is called in a *manifestly constant-evaluated* expression or conversion. That is roughly the case if we call it:

- In a constant expression
- In a constant context (in `if constexpr`, a `consteval` function, or an constant initialization)
- For an initializer of a variable that can be used at compile time

For example:

```
constexpr bool isConstEval() {
  return std::is_constant_evaluated();
}

bool g1 = isConstEval();                    // true
const bool g2 = isConstEval();              // true
static bool g3 = isConstEval();             // true
static int g4 = g1 + isConstEval();         // false

int main()
{
  bool l1 = isConstEval();                  // false
  const bool l2 = isConstEval();            // true
  static bool l3 = isConstEval();           // true
  int l4 = g1 + isConstEval();              // false
```

```
    const int l5 = g1 + isConstEval();            // false
    static int l6 = g1 + isConstEval();           // false
    int l7 = isConstEval() + isConstEval();       // false
    const auto l8 = isConstEval() + 42 + isConstEval();   // true
  }
```

We would get the same effect if `isConstEval()` is called indirectly via a `constexpr` function.

`std::is_constant_evaluated()` with `constexpr` and `consteval`

For example, assume we have the following functions defined:

```
  bool runtimeFunc() {
    return std::is_constant_evaluated(); // always false
  }
  constexpr bool constexprFunc() {
    return std::is_constant_evaluated(); // may be false or true
  }
  consteval bool constevalFunc() {
    return std::is_constant_evaluated(); // always true
  }
```

Then we have the following behavior:

```
  void foo()
  {
    bool b1 = runtimeFunc();            // false
    bool b2 = constexprFunc();          // false
    bool b3 = constevalFunc();          // true

    static bool sb1 = runtimeFunc();     // false
    static bool sb2 = constexprFunc();   // true
    static bool ab3 = constevalFunc();   // true
    const bool cb1 = runtimeFunc();      // ERROR
    const bool cb2 = constexprFunc();    // true
    const bool cb3 = constevalFunc();    // true

    int y = 42;
    static bool sb4 = y + runtimeFunc();     // function yields false
    static bool sb5 = y + constexprFunc();   // function yields false
    static bool ab6 = y + constevalFunc();   // function yields true
    const bool cb4 = y + runtimeFunc();      // function yields false
    const bool cb5 = y + constexprFunc();    // function yields false
    const bool cb6 = y + constevalFunc();    // function yields true
  }
```

By using `constexpr` or `static constinit` instead of `const`, an initialization without `y` would have the same effect as shown for `cb1`, `cb2`, and `cb3`. However, with `y` involved, using `constexpr` or `static constinit` always results in an error because a runtime value is involved.

In general, it makes no sense to use `std::is_constant_evaluated()` in the following situations:

- As a condition in a compile-time `if` because that always yields `true`:

  ```
  if constexpr (std::is_constant_evaluated()) {    // always true
    ...
  }
  ```

  Inside `if constexpr` we have a compile-time context, meaning that the answer to the question of whether we are in a compile-time context is always `true` (regardless of the context the whole function was called from).
- Inside a pure runtime function, because that usually yields `false`.
  The only exception is if `is_constant_evaluated()` is used in a local constant evaluation:

  ```
  void foo() {
    if (std::is_constant_evaluated()) {          // always false
      ...
    }
    const bool b = std::is_constant_evaluated();  // true
  }
  ```
- Inside a `consteval` function, because that always yields `true`:

  ```
  consteval void foo() {
    if (std::is_constant_evaluated()) {          // always true
      ...
    }
  }
  ```

Therefore, using `std::is_constant_evaluated()` usually only makes sense in `constexpr` functions. It also makes no sense to use `std::is_constant_evaluated()` inside a `constexpr` function to call a `consteval` function, because calling a `consteval` function from a `constexpr` function is not allowed in general:[4]

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  if (std::is_constant_evaluated()) {
    return funcConstEval(i);  // ERROR
  }
  else {
    return funcRuntime(i);
  }
}
```

[4] C++23 will probably enable code like this with `if consteval`.

std::is_constant_evaluated() and Operator ?:

In the C++20 standard, there is an interesting example to clarify the way std::is_constant_evaluated() can be used. Slightly modified, it looks as follows:

```
int sz = 10;
constexpr bool sz1 = std::is_constant_evaluated() ? 20 : sz;   // true, so 20
constexpr bool sz2 = std::is_constant_evaluated() ? sz : 20;   // false, so ERROR
```

The reason for this behavior is as follows:[5]

- The initialization of sz1 and sz2 is either static initialization or dynamic initialization.
- For static initialization, the initializer must be constant. Therefore, the compiler attempts to evaluate the initializer with std::is_constant_evaluated() treated as a constant of value true.
  - With sz1, that succeeds. The result is 1, and that is a constant. Therefore, sz1 is a constant initialized with 20.
  - With sz2, the result is sz, which is not a constant. Therefore, sz2 is (notionally) dynamically initialized. The previous result is therefore discarded and the initializer is evaluated with std::is_constant_evaluated() producing false instead. Therefore, the expression to initialize sz2 is also 20.

    However, sz2 is not necessarily a constant because std::is_constant_evaluated() is not necessarily a constant expression during this evaluation. Therefore, the initialization of sz2 with this 20 does not compile.

Using const instead of constexpr makes the situation even more tricky:

```
int sz = 10;
const bool sz1 = std::is_constant_evaluated() ? 20 : sz;   // true, so 20
const bool sz2 = std::is_constant_evaluated() ? sz : 20;   // false, so also 20

double arr1[sz1];   // OK
double arr2[sz2];   // may or may not compile
```

Only sz1 is a compile-time constant and can always be used to initialize an array. For the reason described above, sz2 is also initialized to 20. However, because the initial values is not necessarily a constant, the initialization of arr2 may or may not compile (depending on the compiler and optimizations used).

[5] Thanks to David Vandevoorde for pointing this out.

18.5 Using Heap Memory, Vectors, and Strings at Compile Time

Since C++20, compile-time functions can allocate memory provided the memory is also released at compile time. For this reason, you can now use strings or vectors at compile time. However, there is an important constraint: the strings or vectors created at compile time cannot be used at runtime. The reason is that memory allocated at compile time also has to be released at compile time.

18.5.1 Using Vectors at Compile Time

Here is a first example, using a `std::vector<>` at compile time:

comptime/vector.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <numeric>

template<std::ranges::input_range T>
constexpr auto modifiedAvg(const T& rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};
  // perform several modifications:
  v.push_back(elemType{});
  std::ranges::sort(v);
  auto newEnd = std::unique(v.begin(), v.end());
  ...

  // return average of modified vector:
  auto sum = std::accumulate(v.begin(), newEnd,
                             elemType{});
  return sum / static_cast<double>(v.size());
}
```

Here, we define `modifiedAvg()` with `constexpr` so that it could be called at compile time. Inside, we use the `std::vector<> v`, initialized with the elements of the passed range. This allows us to use the full API of a vector (especially inserting and removing elements). As an example, we insert an element, sort the elements, and remove consecutive duplicates with `unique()`. All these algorithms are `constexpr` now so that we can use them at compile time.

However, at the end, we do ***not*** return the vector. We only return a value computed with the help of a compile-time vector.

We can call this function at compile time:

comptime/vector.cpp

```
#include "vector.hpp"
#include <iostream>
#include <array>

int main()
{
  constexpr std::array orig{0, 8, 15, 132, 4, 77};

  constexpr auto avg = modifiedAvg(orig);
  std::cout << "average: " << avg << '\n';
}
```

Due to the fact that avg is declared with `constexpr`, `modifiedAvg()` is evaluated at compile time.

We could also declare `modifiedAvg()` with `consteval`, in which case we could pass the argument by value because we do not copy elements at runtime:

```
template<std::ranges::input_range T>
consteval auto modifiedAvg(T rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};
  ...
}
```

However, we still cannot declare and initialize a vector at compile time if it can be used at runtime:

```
int main()
{
  constexpr std::vector orig{0, 8, 15, 132, 4, 77};   // ERROR
  ...
}
```

For the same reason, a compile-time function can only return a vector to the caller when the return value is used at compile time:

comptime/returnvector.cpp

```
#include <vector>

constexpr auto returnVector()
{
  std::vector<int> v{0, 8, 15};
  v.push_back(42);
  ...

  return v;
}

constexpr auto returnVectorSize()
{
  auto coll = returnVector();
  return coll.size();
}

int main()
{
  //constexpr auto coll = returnVector();  // ERROR
  constexpr auto tmp = returnVectorSize();  // OK
  ...
}
```

18.5.2 Returning a Collection at Compile Time

Although you cannot return a compile-time vector so that it can be used at runtime, there is a way to return a collection of elements computed at compile time: you can return a `std::array<>`. The only problem is that you need to know the size of the array because the size cannot be initialized by the size of the vector:

```
std::vector v;
...
std::array<int, v.size()> arr;  // ERROR
```

Code like this ***never*** compiles because `size()` is a runtime value and the declaration of `arr` needs a compile-time value. It does not matter whether this code is evaluated at compile time. For this reason, you have to return a fixed-sized array. For example:

comptime/mergevalues.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <array>

template<std::ranges::input_range T>
consteval auto mergeValues(T rg, auto... vals)
{
  // create compile-time vector from passed range:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));  // merge all passed parameters
  std::ranges::sort(v);          // sort all elements

  // return extended collection as array:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, sz> arr{};
  std::ranges::copy(v, arr.begin());
  return arr;
}
```

We use a vector in a `consteval` function to merge a variadic number of passed arguments with the elements of the passed range and sort them all. However, we return the resulting collection as `std::array` so that it can be passed to a runtime context. For example, the following program works fine with this function:

comptime/mergevalues.cpp

```
#include "mergevalues.hpp"
#include <iostream>
#include <array>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};

  // initialization of sorted extended array:
  auto merged = mergeValues(orig, 42, 4);

  // print elements:
  for(const auto& i : merged) {
    std::cout << i << ' ';
  }
}
```

It has the following output:

```
0 3 4 4 8 15 42 77 132
```

If we do not know the resulting size of the array at compile time, we have to declare the returned array with a maximum size and return the resulting size in addition. The function that merges the values might now look as follows:

comptime/mergevaluessz.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <array>
```

```
template<std::ranges::input_range T>
consteval auto mergeValuesSz(T rg, auto... vals)
{
  // create compile-time vector from passed range:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                                               std::ranges::end(rg)};
  (... , v.push_back(vals));  // merge all passed parameters

  std::ranges::sort(v);       // sort all elements

  // return extended collection as array and its size:
  constexpr auto maxSz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, maxSz> arr{};
  auto res = std::ranges::unique_copy(v, arr.begin());
  return std::pair{arr, res.out - arr.begin()};
}
```

In addition, we use std::ranges::unique_copy() to remove consecutive duplicates after sorting and return both the array and the resulting number of elements.

Note that you should declare `arr` with `{}` to ensure that all values in the array are initialized. Compile-time functions are not allowed to yield uninitialized memory.

We can now use the returned value as follows:

comptime/mergevaluessz.cpp

```
#include "mergevaluessz.hpp"
#include <iostream>
#include <array>
#include <ranges>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};

  // initialization of sorted extended array:
  auto tmp = mergeValuesSz(orig, 42, 4);
  auto merged = std::views::counted(tmp.first.begin(), tmp.second);

  // print elements:
  for(const auto& i : merged) {
    std::cout << i << ' ';
  }
}
```

By using the view adaptor `std::views::counted()`, we can easily combine both the returned array and the returned size of elements to use in the array as a single range.

The output of the program is now:

```
0 3 4 8 15 42 77 132
```

18.5.3 Using Strings at Compile Time

For compile-time strings, all operations are `constexpr` now. Therefore, you can use `std::string` but also any other string types such as `std::u8string` at compile time now.

However, there is again the restriction that you cannot use a compile-time string at runtime. For example:

```
consteval std::string returnString()
{
  std::string s = "Some string from compile time";
  ...
  return s;
}

void useString()
{
  constexpr auto s = returnString();  // ERROR
  ...
}

constexpr void useStringInConstexpr()
{
  std::string s = returnString();     // ERROR
  ...
}

consteval void useStringInConsteval()
{
  std::string s = returnString();     // OK
  ...
}
```

You also cannot solve the problem by returning just the compile-time string with `data()` or `c_str()` or as a `std::string_view`. You would return the address of memory allocated at compile time. Compilers raise a compile-time error if that happens.

However, we can use the same trick as described for vectors above. We can convert the string into an array of fixed size and return both the array and the size of the vector.

Here is a complete example:

comptime/comptimestring.cpp

```
#include <iostream>
#include <string>
#include <array>
#include <cassert>
#include "asstring.hpp"

// function template to export a compile-time string to runtime:
template<int MaxSize>
consteval auto toRuntimeString(std::string s)
{
  // ensure the size of the exported array is large enough:
  assert(s.size() <= MaxSize);

  // create a compile-time array and copy all characters into it:
  std::array<char, MaxSize+1> arr{};   // ensure all elems are initialized
  for (int i = 0; i < s.size(); ++i) {
    arr[i] = s[i];
  }

  // return the compile-time array and the string size:
  return std::pair{arr, s.size()};
}

// function to import an exported compile-time string at runtime:
std::string fromComptimeString(const auto& dataAndSize)
{
  // init string with exported array of chars and size:
  return std::string{dataAndSize.first.data(),
                     dataAndSize.second};
}

// test the functions:
consteval auto comptimeMaxStr()
{
  std::string s = "max int is " + asString(std::numeric_limits<int>::max())
                + " (" + asString(std::numeric_limits<int>::digits + 1)
                + " bits)";

  return toRuntimeString<100>(s);
}
```

```
int main()
{
  std::string s = fromComptimeString(comptimeMaxStr());
  std::cout << s << '\n';
}
```

Again, we define two helper functions to export a compile-time string to a runtime string:

- The compile-time function `toRuntimeString()` converts a string into a `std::array<>` and returns the array and the size of the string:

  ```
  template<int MaxSize>
  consteval auto toRuntimeString(std::string s)
  {
    assert(s.size() <= MaxSize);          // ensure array size fits

    // create a compile-time array and copy all characters into it:
    std::array<char, MaxSize+1> arr{};    // ensure all elems are initialized
    for (int i = 0; i < s.size(); ++i) {
      arr[i] = s[i];
    }

    return std::pair{arr, s.size()};      // return array and size
  }
  ```

  By using `assert()`, we double check that the size of the array is large enough. In the same way, we could double check at compile time that we do not waste too much memory.

- The runtime function `fromComptimeString()` then takes the returned array and size to initialize a runtime string and returns it:

  ```
  std::string fromComptimeString(const auto& dataAndSize)
  {
    return std::string{dataAndSize.first.data(),
                       dataAndSize.second};
  }
  ```

The test case of the function uses the helper function `asString()`, which can be used at compile time and runtime to convert an integral value into a string.

The program has, for example, the following output:

```
max int is 2147483647 (32 bits)
```

18.6 Other constexpr Extensions

Besides the ability to use heap memory at compile time, compile-time functions (whether declared with `constexpr` or `consteval`) can use a couple of additional language features since C++20. As a consequence and in addition, a couple of library features can now also be used at compile time.

18.6.1 constexpr Language Extensions

Since C++20, the following language features can be used in compile-time functions (whether declared with `constexpr` or `consteval`):

- You can now use heap memory at compile time.
- Runtime polymorphism is supported:
  - You can now use `virtual` functions.
  - You can now use `dynamic_cast`.
  - You can now use `typeid`.
- You can have `try`-`catch` blocks now (but you are still not allowed to `throw`).
- You can now change the active member of a `union`.

Note that you are still not allowed to use `static` in `constexpr` or `consteval` functions.

18.6.2 constexpr Library Extensions

The C++ standard library has extended the utilities that can be used at compile time.

constexpr Algorithms and Utilities

Most algorithms in `<algorithm>`, `<numeric>`, and `<utility>` are `constexpr` now. This means that you can now sort and accumulate elements at compile time (see the example using vectors at compile time).

However, parallel algorithms (algorithms that have an execution policy parameter) can still only be used at runtime.

constexpr Library Types

Several types of the C++ standard library now have better support for `constexpr` so that objects can be used (better) at compile time:

- Vectors and strings can be used at compile time now.
- A couple of `std::complex<>` operations became `constexpr`.
- A couple of missing `constexpr` in `std::optional<>` and `std::variant<>` were added.
- `constexpr` was added to `std::invoke()`, `std::ref()`, `std::cref()`, `mem_fn()`, `not_fn()`, `std::bind()`, and `std::bind_front()`,
- In `pointer_traits`, `pointer_to()` for raw pointers can now be used at compile time.

18.7 Afternotes

The keyword `constinit` was first proposed as an attribute by Eric Fiselier in http://wg21.link/p1143r0. The finally accepted wording was formulated by Eric Fiselier in http://wg21.link/p1143r2.

The keyword `consteval` was first proposed by Richard Smith, Andrew Sutton, and Daveed Vandevoorde in http://wg21.link/p1073r0. The finally accepted wording was formulated by Richard Smith, Andrew Sutton, and Daveed Vandevoorde in http://wg21.link/p1073r3. A small fix was later added by David Stone in http://wg21.link/p1937r2.

`std::is_constant_evaluated()` was first proposed as a `constexpr` operator by Daveed Vandevoorde in http://wg21.link/p0595r0. The finally accepted wording was formulated by Richard Smith, Andrew Sutton, and Daveed Vandevoorde in http://wg21.link/p0595r2.

Compile-time containers (such as vectors and strings) were first proposed by Daveed Vandevoorde in http://wg21.link/p0597. The finally accepted language features were formulated by Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, and Daveed Vandevoorde in http://wg21.link/p0784r7. The finally accepted library features were formulated by Antony Polukhin in http://wg21.link/p0858r0 and by Louis Dionne in http://wg21.link/p1004r2. and http://wg21.link/00980r1.

This page is intentionally left blank

Chapter 19

Non-Type Template Parameter (NTTP) Extensions

C++ template parameters do not have to be only types. They can also be values (*non-type template parameters (NTTP)*). However, there are restrictions on the types these values can have. C++20 adds more types that can be used for *non-type template parameters*. Floating-point values, objects of data structures (such as `std::pair<>` and `std::array<>`) and simple classes, and even lambdas can now be passed as template arguments. This chapter describes these types and useful applications of this feature.

Note that there is another new feature for non-type template parameters: since C++20, you can use concepts to constrain the values of NTTPs.

19.1 New Types for Non-Type Template Parameters

Since C++20, you can use new types for non-type template parameters:

- Floating-point types (such as `double`)
- Structures and simple classes (such as `std::pair<>`), which indirectly also allows you to use string literals as template parameters
- Lambdas

In fact, non-type template parameters may now be all ***structural types***. A *structural type* is a type that

- either is a (`const` or `volatile` qualified) arithmetic, enum, or pointer type
- or and lvalue reference type
- or a literal type (is either an aggregate or has a `constexpr` constructor, no copy/move constructor, no destructor, no copy/move constructor or destructor, and where every initialization of data members is a constant expression) where:
  - All non-static members are public and not `mutable` and use only structural types or arrays thereof
  - All base classes (if there are any) are inherited publicly and also structural types

Let us look at what the consequences of this new definition are.

19.1.1 Floating-Point Values as Non-Type Template Parameters

Consider the following example:

lang/nttpdouble.cpp

```
#include <iostream>
#include <cmath>

template<double Vat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + Vat)));
}

int main()
{
  std::cout << addTax<0.19>(100) << '\n';
  std::cout << addTax<0.19>(4199) << '\n';
  std::cout << addTax<0.07>(1950) << '\n';
}
```

The output of the program is as follows:

```
119
4997
2087
```

By declaring `addTax()` as follows:

```
template<double Vat>
int addTax(int value)
```

the function template `addTax()` takes a `double` as a template parameter, which is then used as value-added tax to add it to an integral value.

Passing a floating-point value is now also allowed when the non-type template parameter is declared with `auto`:

```
template<auto Vat>
int addTax(int value)
{
  ...
}

std::cout << addTax<0>(1950) << '\n';      // Vat is the int value 0
std::cout << addTax<0.07>(1950) << '\n';   // Vat is the double value 0.07
```

In the same way, you can now use floating-point values in class templates (declared as `double` or as `auto`):

```
template<double Vat>
class Tax {
  ...
};
```

Dealing with Inexact Floating-Point Values

Due to rounding errors, values of floating-point types end up being slightly imprecise. This has an effect when dealing with floating-point values as template parameters. The issue is, when two instantiations of a template have the same type.

Consider the following example:

lang/nttpdouble2.cpp

```
#include <iostream>
#include <limits>
#include <type_traits>

template<double Val>
class MyClass {
};

int main()
{
  std::cout << std::boolalpha;
  std::cout << std::is_same_v<MyClass<42.0>, MyClass<17.7>>        // always false
            << '\n';
  std::cout << std::is_same_v<MyClass<42.0>, MyClass<126.0 / 3>>   // true or false
            << '\n';
  std::cout << std::is_same_v<MyClass<42.7>, MyClass<128.1/ 3>>    // true or false
            << "\n\n";

  std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
                              MyClass<0.3 + 0.1 + 0.00001>>        // true or false
            << '\n';

  std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
                              MyClass<0.00001 + 0.3 + 0.1>>        // true or false
            << "\n\n";

  constexpr double NaN = std::numeric_limits<double>::quiet_NaN();
  std::cout << std::is_same_v<MyClass<NaN>, MyClass<NaN>>          // always true
            << '\n';
}
```

The output of this program depends on the platform. Often, it is as follows:

```
false
true
false
---
true
false
---
true
```

Note that templates instantiated for `NaN` always have the same type even though `NaN` == `NaN` is `false`.

19.1.2 Objects as Non-Type Template Parameters

Since C++20, you can use an object/value of a data structure or class as a non-type template parameter provided all members are public and the type is a literal type.

Consider the following example:

lang/nttpstruct.cpp

```cpp
#include <iostream>
#include <cmath>
#include <cassert>

struct Tax {
  double value;

  constexpr Tax(double v)
   : value{v} {
      assert(v >= 0 && v < 1);
  }

  friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
    return strm << t.value;
  }
};

template<Tax Vat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + Vat.value)));
}

int main()
{
  constexpr Tax tax{0.19};
  std::cout << "tax: " << tax << '\n';

  std::cout << addTax<tax>(100) << '\n';
  std::cout << addTax<tax>(4199) << '\n';
  std::cout << addTax<Tax{0.07}>(1950) << '\n';
}
```

Here, we declare a literal data structure `Tax` with public members, a `constexpr` constructor, and an additional member function:

```
struct Tax {
  double value;

  constexpr Tax(double v) {
    ...
  }
  friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
    ...
  }
};
```

This allows us to pass objects of this type as template arguments:

```
constexpr Tax tax{0.19};
std::cout << "tax: " << tax << '\n';
std::cout << addTax<tax>(100) << '\n';    // pass Tax object as a template argument
```

This works if the data structure or class is a ***structural type***. This roughly means that:

- All non-static members are public and not `mutable` and use only structural types or arrays thereof
- All base classes (if there are any) are inherited publicly and also structural types
- The type is a literal type (is either an aggregate or has a `constexpr` constructor, no copy/move constructor, no destructor, no copy/move constructor or destructor, and where every initialization of data members is a constant expression)

For example:

lang/nttpstruct2.cpp

```
#include <iostream>
#include <array>

constexpr int foo()
{
  return 42;
}

```

```
struct Lit {
  int x = foo();          // OK because foo() is constexpr
  int y;
  constexpr Lit(int i)    // OK because constexpr
   : y{i} {
  }
};

struct Data {
  int i;
  std::array<double,5> vals;
  Lit lit;
};

template<auto Obj>
void func()
{
  std::cout << typeid(Obj).name() << '\n';
}

int main()
{
  func<Data{42, {1, 2, 3}, 42}>();     // OK

  constexpr Data d2{1, {2}, 3};
  func<d2>();
}
```

Type Type cannot be used if its constructor or `foo()` would not be `constexpr` or a `std::string` member would be used.

`std::pair<>` and `std::array<>` Values as Non-Type Template Parameters

As a consequence, you can now use compile-time objects of types `std::pair<>` and `std::array<>` as template parameters:[1]

```
template<auto Val>
class MyClass {
  ...
};

MyClass<std::pair{47,11}> mcp;          // OK since C++20
MyClass<std::array{0, 8, 15}> mca;      // OK since C++20
```

[1] For this, C++ had to add the additional requirement that `std::pair<>` and `std::array<>` may not be implemented with a private base class, which some implementers did before C++20 (see `http://wg21.link/lwg3382`).

Strings as Non-Type Template Parameters

Note that a data structure that has a character array as a public member is a structural type. That way, we can now quite easily pass string literals as template arguments.

For example:

lang/nttpstring.cpp

```
#include <iostream>
#include <string_view>

template<auto Prefix>
class Logger {
  ...
 public:
  void log(std::string_view msg) const {
    std::cout << Prefix << msg << '\n';
  }
};

template<std::size_t N>
struct Str {
  char chars[N];
  const char* value() {
    return chars;
  }
  friend std::ostream& operator<< (std::ostream& strm, const Str& s) {
    return strm << s.chars;
  }
};
template<std::size_t N> Str(const char(&)[N]) -> Str<N>;   // deduction guide

int main()
{
  Logger<Str{"> "}> logger;
  logger.log("hello");
}
```

The program has the following output:

```
> hello
```

19.1.3 Lambdas as Non-Type Template Parameters

Because lambdas are just shortcuts for function objects, they can now also be used as non-type template parameters provided the lambda can be used at compile time.

Consider the following example:

lang/nttplambda.cpp

```
#include <iostream>
#include <cmath>

template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

int main()
{
  auto getDefaultTax = [] {
    return 0.19;
  };

  std::cout << addTax<getDefaultTax>(100) << '\n';
  std::cout << addTax<getDefaultTax>(4199) << '\n';
  std::cout << addTax<getDefaultTax>(1950) << '\n';
}
```

The function template addTax() uses a helper function, which can now also be a lambda:

```
template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

We can now pass a lambda to this function template:

```
auto getDefaultTax = [] {
  return 0.19;
};

addTax<getDefaultTax>(100)            // passes lambda as template argument
```

We could even define the lambda directly when calling the function template:

```
addTax<[]{ return 0.19; }>(100)    // passes lambda as template argument
```

Note that it is a good idea to constrain the template parameter with the concepts std::invocable or std::regular_invocable. That way, you can document and ensure that the passed callable can be called with arguments of the specified types.

By using just std::invocable auto, we require that the callable takes no arguments. If the passed callable should take arguments, you need something like this:

```
template<std::invocable<std::string> auto GetVat>
int addTax(int value, const std::string& name)
{
  double vat = GetVat(name);   // get VAT according to the passed name
  ...
}
```

Note that you cannot skip auto in the declaration of the function template. We use std::invocable as a type constraint for the type a passed ***value/object/callback*** has:

```
template<std::invocable auto GetVat>   // GetVat is a callable with constrained type
```

Without auto, we would declare a function template that has an ordinary ***type*** parameter, which we constrain:

```
template<std::invocable GetVat>        // GetVat is a constrained type
```

It would also work if we declare the function template with the concrete type of the lambda (which, however, means that we have to define the lambda first):

```
auto getDefaultTax = [] {
  return 0.19;
};

template<decltype(getDefaultTax) GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

Note the following constraints for using lambdas as non-type template parameters:

- The lambda may not capture anything.
- In must be possible to use the lambda at compile time.

Fortunately, since C++17, any lambda is implicitly constexpr provided it uses only features that are valid for compile-time computing. Alternatively, you can declare the lambda with constexpr or consteval to get an error on the lambda itself instead of when using it as a template parameter if invalid compile-time features are used.

19.2 Afternotes

The request for more non-type template parameters has existed since the first C++ standard. We have already discussed it in the first edition of *C++ Templates – The Complete Guide* (see `http://tmplbook.com`).

Allowing arbitrary literal types for non-type template parameters was first proposed by Jens Maurer in `http://wg21.link/n3413`. Allowing class objects as non-type template parameters was then proposed for C++20 by Jeff Snyder in `http://wg21.link/p0732r0`. Allowing floating-point values as non-type template parameters was then proposed for C++20 by Jorg Brown in `http://wg21.link/p1714r0`.

The finally accepted wordings were formulated by Jeff Snyder and Louis Dionne in `http://wg21.link/p0732r2` and by Jens Maurer in `http://wg21.link/p1907r1`.

Chapter 20

New Type Traits

This chapter presents several type traits (and two low-level functions for types) that C++20 introduces in its standard library.

Table *New type traits* lists these new type traits of C++20 (all are defined in the namespace `std`).

Trait	Effect
`is_bounded_array_v<`*T*`>`	Yields true if type *T* is an array type with known extent
`is_unbounded_array_v<`*T*`>`	Yields true if type *T* is an array type with unknown extent
`is_nothrow_convertible_v<`*T*, *T2*`>`	Yields true if *T* is convertible to type *T2* without throwing
`is_layout_compatible_v<`*T1*, *T2*`>`	Yields true if *T1* is layout-compatible with type *T2*
`is_pointer_interconvertible...` `_base_of_v<`*BaseT*, *DerT*`>`	Yields true if a pointer to *DerT* can safely be converted to a pointer to its base type *BaseT*
`remove_cvref_t<`*T*`>`	Yields type *T* without reference, top-level `const`, `volatile`
`unwrap_reference_t<`*T*`>`	Yields the wrapped type of type `T` if it is a `std::reference_wrapper<>` or otherwise `T`
`unwrap_ref_decay_t<`*T*`>`	Yields the wrapped type of type `T` if it is a `std::reference_wrapper<>` or otherwise its decayed type
`common_reference_t<`*T...*`>`	Yields the common type of all types *T...* to which you can assign a value
`type_identity_t<`*T*`>`	Yields type *T* as it is
`iter_difference_t<`*T*`>`	Yields the difference type of the incrementable/iterator type *T*
`iter_value_t<`*T*`>`	Yields the value/element type of the pointer/iterator type *T*
`iter_reference_t<`*T*`>`	Yields the reference type of the pointer/iterator type *T*
`iter_rvalue_reference_t<`*T*`>`	Yields the rvalue reference type of the pointer/iterator type *T*
`is_clock_v<`*T*`>`	Yields true if *T* is a clock type
`compare_three_way_result_t<`*T*`>`	Yields the type of comparing two values with operator `<=>`

Table 20.1. New type traits

The following sections discuss these traits in detail, except for the following traits:

- `std::is_clock_v<>` is discussed in the section about new chrono features.
- `std::compare_three_way_result_t<>` is discussed in the section about the new three-way comparisons.

Note that in addition, the type trait `std::is_pod<>` was deprecated with C++20.

20.1 New Type Traits for Type Classification

20.1.1 `is_bounded_array_v<>` and `is_unbounded_array_v`

`std::`**`is_bounded_array_v`**`<`*`T`*`>`
`std::`**`is_unbounded_array_v`**`<`*`T`*`>`

yield whether type *T* is a bounded/unbounded array (extent known/unknown).
For example:

```
int a[5];
std::is_bounded_array_v<decltype(a)>      // true
std::is_unbounded_array_v<decltype(a)>    // false

extern int b[];
std::is_bounded_array_v<decltype(b)>      // false
std::is_unbounded_array_v<decltype(b)>    // true
```

20.2 New Type Traits for Type Inspection

20.2.1 `is_nothrow_convertible_v<>`

`std::`**`is_nothrow_convertible_v`**`<`*`From`*`, `*`To`*`>`

yields whether type *From* is convertible to type To with the guarantee not to throw any exception.
For example:

```
// char* to std::string:
std::is_convertible_v<char*, std::string>                       // true
std::is_nothrow_convertible_v<char*, std::string>               // false

// std::string to std::string_view:
std::is_convertible_v<std::string, std::string_view>            // true
std::is_nothrow_convertible_v<std::string, std::string_view>    // true
```

20.3 New Type Traits for Type Conversion

20.3.1 `remove_cvref_t<>`

`std::`**`remove_cvref_t`**`<`*T*`>`

yields type `T` without being a reference, top-level `const`, or `volatile`.

The expression

```
std::remove_cvref_t<T>
```

is equivalent to:

```
std::remove_cv_t<remove_reference_t<T>>.
```

For example:

```
std::remove_cvref_t<const std::string&>  //std::string
std::remove_cvref_t<const char* const>   //const char*
```

20.3.2 `unwrap_reference<>` and `unwrap_ref_decay_t`

`std::`**`unwrap_reference_t`**`<`*T*`>`

yields the wrapped type of `T` if it is a `std::reference_wrapper<>` (created with `std::ref()` or `std::cref()`) or otherwise `T`.

`std::`**`unwrap_ref_decay_t`**`<`*T*`>`

yields the wrapped type of `T` if it is a `std::reference_wrapper<>` (created with `std::ref()` or `std::cref()`) or otherwise the decayed type of `T`.

For example:

```
std::unwrap_reference_t<decltype(std::ref(s))>    //std::string&
std::unwrap_reference_t<decltype(std::cref(s))>   //const std::string&
std::unwrap_reference_t<decltype(s)>              //std::string
std::unwrap_reference_t<decltype(s)&>             //std::string&
std::unwrap_reference_t<int[4]>                   //int[4]

std::unwrap_ref_decay_t<decltype(std::ref(s))>    //std::string&
std::unwrap_ref_decay_t<decltype(std::cref(s))>   //const std::string&
std::unwrap_ref_decay_t<decltype(s)>              //std::string
std::unwrap_ref_decay_t<decltype(s)&>             //std::string
std::unwrap_ref_decay_t<int[4]>                   //int*
```

20.3.3 `common_reference<>_t`

`std::`**`common_reference_t`**`<`*T...*`>`

yields the common type (if there is one) of all types *T...* to which you can assign a value. Therefore, given types `T1`, `T2`, and `T3`, the trait yields the type where you can assign values of all three types. Ideally, it is a reference type. However, if a type conversion is involved and that creates a temporary object, it is a value type.

For example:

```
std::common_reference_t<int&, int>                             // int
std::common_reference_t<int&, int&>                            // int&
std::common_reference_t<int&, int&&>                           // const int&
std::common_reference_t<int&&, int&&>                          // int&&
std::common_reference_t<int&, double>                          // double
std::common_reference_t<int&, double&&>                        // double
std::common_reference_t<char*, std::string, std::string_view> // std::string_view
std::common_reference_t<char, std::string>                     // ERROR
```

20.3.4 `type_identity_t<>`

`std::`**`type_identity_t`**`<`*`T`*`>`

yields just type `T`.

This type trait has a surprising number of use cases:

- You can disable the fact that a parameter is used to deduce a template parameter. For example:

  ```
  template<typename T>
  void insert(std::vector<T>& coll, const std::type_identity_t<T>& value)
  {
     coll.push_back(value);
  }

  std::vector<double> coll;
  ...
  insert(coll, 42);      // OK: type of 42 not used to deduce type T
  ```

  If the parameter `value` were to be declared with just `const T&`, the compiler would raise an error because it would deduce two different types for type `T`.

- You can use it as a building block to define type traits that yield types. For example, you could define a type trait that removes constness simply as follows:[1]

  ```
  template<typename T>
  struct remove_const : std::type_identity<T> {
  };

  template<typename T>
  struct remove_const<const T> : std::type_identity<T> {
  };
  ```

[1] See the talk *Modern Template Metaprogramming: A Compendium* by Walter E. Brown at CppCon 2014 (`http://www.youtube.com/watch?v=Am2is2QCvxY`) for a source of this example.

20.4 New Type Traits for Iterators

This section lists the traits for iterators as listed in the section about basic type functions for iterators.

20.4.1 `iter_difference_t<>`

`std::`**`iter_difference_t`**`<`*`T`*`>`

yields the difference type that corresponds to the incrementable/iterator type `T`.

The trait is especially provided to deal with the value type of two objects of an indirectly readable type. In contrast to the traditional iterator traits (`std::iterator_traits<>`), this trait can deal correctly with the new iterator categories.

Note that there is no corresponding data structure `std::iter_difference` with a member named `type`. Instead, the type trait is defined by trying to use the member `difference_type` of the new auxiliary type `std::incrementable_traits<>`, which is already defined as follows:

- If specialized, `std::incrementable_traits<T>::difference_type` is used.
- For raw pointers, `std::ptrdiff_t` is used.
- Otherwise, if defined, `T::difference_type` is used.
- Otherwise, the signed integral difference type of the difference between two `T`s is used.
- For a `const T`, the difference type of `T` is used.

For example:

```
using T1 = std::iter_difference_t<int*>;                      // std::ptrdiff_t
using T2 = std::iter_difference_t<std::string>;               // std::ptrdiff_t
using T3 = std::iter_difference_t<std::vector<long>>;         // std::ptrdiff_t
using T4 = std::iter_difference_t<int>;                       // int
using T5 = std::iter_difference_t<std::chrono::sys_seconds>;  // ERROR
```

20.4.2 `iter_value_t<>`

`std::`**`iter_value_t`**`<`*`T`*`>`

yields the non-`const` value/element type that corresponds to the pointer/iterator type `T`.

The trait is especially provided to deal with the value type of an indirectly readable type. In contrast to the traditional iterator traits (`std::iterator_traits<>`), this trait can deal correctly with the new iterator categories.

Note that there is no corresponding data structure `std::iter_value` with a member named `type`. Instead, the type trait is defined by trying to use the member `value_type` of the new auxiliary type `std::indirectly_readable_traits<>`, which is already defined as follows:

- If specialized, `std::indirectly_readable_traits<T>::value_type` is used.
- For raw pointers, the non-`const`/`volatile` type it refers to is used.
- Otherwise, if defined, `remove_cv_t<T::value_type>` is used.
- Otherwise, if defined, `remove_cv_t<T::element_type>` is used.
- For a `const T`, the value type of `T` is used.

For example:

```
using T1 = std::iter_value_t<int*>;                          // int
using T2 = std::iter_value_t<const int* const>;              // int
using T3 = std::iter_value_t<std::string>;                   // char
using T4 = std::iter_value_t<std::vector<long>>;             // long
using T5 = std::iter_value_t<int>;                           // ERROR
```

20.4.3 `iter_reference_t<>` and `iter_rvalue_reference_t<>`

std::**iter_reference_t**<*T*>

yields the lvalue reference type that corresponds to the dereferenceable pointer/iterator type `T`. It is equivalent to

```
decltype(*declval<T&>())
```

std::**iter_rvalue_reference_t**<*T*>

yields the rvalue value type that corresponds to the dereferenceable pointer/iterator type `T`. It is equivalent to

```
decltype(std::ranges::iter_move(declval<T&>()))
```

The traits are especially provided to deal with the value type of an indirectly writable type. In contrast to the traditional iterator traits (`std::iterator_traits<>`), these traits can deal correctly with the new iterator categories.

Note that there is no corresponding data structure `std::iter_value` with a `type` member. Instead, the traits are defined directly as described above.

For example:

```
using T1 = std::iter_reference_t<int*>;                      // int&
using T2 = std::iter_reference_t<const int* const>;          // const int&
using T3 = std::iter_reference_t<std::string>;               // ERROR
using T4 = std::iter_reference_t<std::vector<long>>;         // ERROR
using T5 = std::iter_reference_t<int>;                       // ERROR

using T6 = std::iter_rvalue_reference_t<int*>;               // int&&
using T7 = std::iter_rvalue_reference_t<const int* const>;   // const int&&
using T8 = std::iter_rvalue_reference_t<std::string>;        // ERROR
```

20.5 Type Traits and Functions for Layout Compatibility

In a couple of situations it is important to know whether two types or pointers to types can safely be converted to each other. The C++ standard uses the term ***layout-compatible*** for that.

To check the layout-compatible relationships between class members, C++20 also introduces two new ordinary functions. They have the benefit that they can be used in runtime contexts.

20.5.1 `is_layout_compatible_v<>`

`std::`**`is_layout_compatible_v`**`<`*`T1`*`, `*`T2`*`>`

yields whether types *T1* and *T2* are *layout-compatible* so that you can safely convert pointers to them with `reinterpret_cast`.

For example:

```
struct Data {
  int i;
  const std::string s;
};

class Type {
 private:
  const int id = nextId();
  std::string name;
 public:
  ...
};

std::is_layout_compatible_v<Data, Type>            // true
```

Note that for *layout-compatibility* it is not enough that the type and the bits fit roughly. According to the language rules:

- Signed types are never layout-compatible to unsigned types.
- `char` is even never layout-compatible to both `signed char` and `unsigned char`.
- References are never layout-compatible to non-references.
- Arrays of different (even layout-compatible) types are never layout-compatible.
- Enumeration types are never layout-compatible to their underlying type.

For example:

```
enum class E {};
enum class F : int {};

std::is_layout_compatible_v<E, F>                  // true
std::is_layout_compatible_v<E[2], F[2]>            // false
std::is_layout_compatible_v<E, int>                // false
```

```
std::is_layout_compatible_v<char, char>           // true
std::is_layout_compatible_v<char, signed char>    // false
std::is_layout_compatible_v<char, unsigned char>  // false
std::is_layout_compatible_v<char, char&>          // false
```

20.5.2 `is_pointer_interconvertible_base_of_v<>`

std::**is_pointer_interconvertible_base_of**<*Base*, *Der*>

yields true if a pointer to type *Der* can safely be converted to a pointer to its base type *Base* with `reinterpret_cast`.

If both have the same type, the trait always yields `true`.

For example:

```
struct B1 { };
struct D1 : B1 { int x; };

struct B2 { int x; };
struct D2 : B2 { int y; };    // no standard-layout type

std::is_pointer_interconvertible_base_of_v<B1, D1>    // true
std::is_pointer_interconvertible_base_of_v<B2, D2>    // false
```

The reason that a pointer to `D2` cannot be safely converted into a pointer to `B2` is that it is not a standard-layout type because not all members are defined in the same class with the same access.

20.5.3 `is_corresponding_member()`

```
template<typename S1, typename S2, typename M1, typename M2>
constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;
```

returns whether `m1` and `m2` point to layout-compatible members of `S1` and `S2`. This means that these members and all members in front of these members have to be layout-compatible. The function yields `true` if and only if `S1` and `S2` are standard-layout types, `M1` and `M2` are object types, and `m1` and `m2` are not null.

For example:

```
struct Point2D { int a; int b; };
struct Point3D { int x; int y; int z; };
struct Type1 { const int id; int val; std::string name; };
struct Type2 { unsigned int id; int val; };

std::is_corresponding_member(&Point2D::b, &Point3D::y)   // true
std::is_corresponding_member(&Point2D::b, &Point3D::z)   // false (2nd versus 3rd int)
std::is_corresponding_member(&Point2D::b, &Type1::val)   // true
std::is_corresponding_member(&Point2D::b, &Type2::val)   // false (signed vs. unsigned)
```

20.5.4 `is_pointer_interconvertible_with_class()`

```
template<typename S, typename M>
constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;
```

returns whether each object `s` of type `S` is *pointer-interconvertible* with its subobject `s.*m`. The function yields `true` if and only if `S` is a standard-layout type, `M` is an object type, and `m` is not null.

For example:

```
struct B1 { int x; };
struct B2 { int y; };

struct DB1 : B1 {};
struct DB1B2 : B1, B2 {};    // not a standard-layout type

std::is_pointer_interconvertible_with_class<B1, int>(&DB1::x)       // true
std::is_pointer_interconvertible_with_class<DB1, int>(&B1::x)       // true
std::is_pointer_interconvertible_with_class<DB1, int>(&DB1::x)      // true

std::is_pointer_interconvertible_with_class<B1, int>(&DB1B2::x)     // true
std::is_pointer_interconvertible_with_class<DB1B2, int>(&B1::x)     // false
std::is_pointer_interconvertible_with_class<DB1B2, int>(&DB1B2::x)  // false
```

The C++20 standard explains the reason for the last two statements being `false` in a note:

> The type of a pointer-to-member expression `&C::b` is not always a pointer to a member of C. This leads to potentially surprising results when using these functions in conjunction with inheritance:
>
> ```
> struct A { int a; };                // a standard-layout class
> struct B { int b; };                // a standard-layout class
> struct C: public A, public B { };   // not a standard-layout class
>
> std::is_pointer_interconvertible_with_class(&C::b)      // true
>   // true because, despite its appearance, &C::b has type
>   // "pointer to member of B of type int"
>
> std::is_pointer_interconvertible_with_class<C>(&C::b)   // false
>   // false because it forces the use of class C and fails
> ```

20.6 Afternotes

The type traits `is_bounded_array<>` and `is_unbounded_array<>` were accepted as proposed by Walter E. Brown and Glen J. Fernandes in http://wg21.link/p1357r1.

The type trait `is_nothrow_convertible<>` was accepted as proposed by Daniel Krügler in http://wg21.link/p0758r1.

The type trait `common _reference<>` was accepted as part of the ranges library by Eric Niebler, Casey Carter, and Christopher Di Bella in `http://wg21.link/p0896r4`.

The type traits `unwrap _reference<>` and `unwrap_ref_decay<>` were accepted as proposed by Vicente J. Botet Escriba in `http://wg21.link/p0318r1`.

The type trait `remove_cvref<>` was accepted as proposed by Walter E. Brown in `http://wg21.link/p0550r2`.

The type trait `type_identity<>` was accepted as proposed by Timur Doumler in `http://wg21.link/p0887r1`.

The iterator type traits were accepted as part of adopting the ranges library as proposed by Eric Niebler, Casey Carter, and Christopher Di Bella in `http://wg21.link/p0896r4`.

The type traits `is_layout_compatible<>` and `is_pointer_interconvertible_base_of<>` as well as the functions `is_corresponding_member()` and `is_pointer_interconvertible_with_class()` were accepted as proposed by Lisa Lippincott in `http://wg21.link/p0466r5`.

Chapter 21

Small Improvements for the Core Language

This chapter presents additional features and extensions that C++20 introduces for its core language that have not been covered in this book yet.

Additional small features for generic programming are described in the next chapter.

21.1 Range-Based `for` Loop with Initialization

C++17 introduced optional initialization for the `if` and `switch` control structures. C++20 now introduces such an optional initialization for the range-based `for` loop.

For example, the following code iterates over elements of a collection while incrementing a counter:

```
for (int i = 1; const auto& elem : coll) {
  std::cout << std::format("{:3}: {}\n", i, elem);
  ++i;
}
```

As another example, consider this code that iterates over the entries of the directory `dirname`:

```
for (std::filesystem::path p{dirname};
     const auto& e : std::filesystem::directory_iterator{p}) {
  std::cout << " " << e.path().lexically_normal().string() << '\n';
}
```

As a third example, the following code locks a mutex while iterating over a collection:

```
for (std::lock_guard lg{collMx}; const auto& elem : coll) {
  std::cout << elem: << elem << '\n';
}
```

Note that the usual caveats for initializers in control structures apply: the initializer needs to declare a variable with a ***name***. Otherwise, the initialization itself is an expression that creates and immediately destroys a temporary object. As a consequence, initializing a lock guard without a name is a logical error because the guard would no longer lock when the iteration happens:

```
for (std::lock_guard{collMx}; const auto& elem : coll) {   // runtime ERROR
  std::cout << elem: << elem << '\n';                      // - no longer locked
}
```

The range-based `for` loop with initialization can also be used as a workaround for a bug in the range-based `for` loop. According to its specification, using the range-based `for` loop might result in a (fatal) runtime error when iterating over a reference to a temporary object.[1] For example:

```
std::optional<std::vector<int>> getValues();              // forward declaration

for (int i : getValues().value()) {                       // fatal runtime ERROR
  ...
}
```

Using the range-based `for` loop with initialization avoids this problem:

```
std::optional<std::vector<int>> getValues();              // forward declaration

for (auto&& optColl = getValues(); int i : optColl) {     // OK
  ...
}
```

In the same way, you can fix a broken iteration using a span:

```
for (auto elem : std::span{getCollOfConst()}) ...              // fatal runtime error

for (auto&& coll = getCollOfConst(); auto elem : std::span{coll}) ...      // OK
```

21.2 `using` for Enumeration Values

Assume we have a scoped enumeration type (declared with `enum class`):

```
enum class Status{open, progress, done = 9};
```

In contrast to unscoped enumeration types (`enum` without `class`), the values of this type need a qualification with their type name:

```
auto x = Status::open;   // OK
auto x = open;           // ERROR
```

However, qualifying each value all the time can become a bit tedious in some contexts where it is clear that we have no conflicts. To make the use of scoped enumeration types more convenient, you can now use a *using enum declaration*.

[1] The problem has been known since 2009 (see `http://wg21.link/cwg900`). However, the C++ standards committee has so far not been willing to fix this bug as proposed, for example, in `http://wg21.link/p2012`.

A typical example is a switch over all possible enumeration values. You can now implement it as follows:

```
void print(Status s)
{
  switch (s) {
   using enum Status;    // make enum values available in current scope
   case open:
    std::cout << "open";
    break;
   case progress:
    std::cout << "in progress";
    break;
   case done:
    std::cout << "done";
    break;
  }
}
```

As long as no other symbol with the name open, progress, or done is declared in the scope of print(), this code works fine.

You can also use multiple using declarations for specific enumeration values now:

```
void print(Status s)
{
  switch (s) {
   using Status::open, Status::progress, Status::done;
   case open:
    std::cout << "open";
    break;
   case progress:
    std::cout << "in progress";
    break;
   case done:
    std::cout << "done";
    break;
  }
}
```

That way, you know exactly which names you make available in the current scope.

Note that you can also use the using declaration for unscoped enumeration types. It is not necessary, but that way, you do not have to know how an enumeration type is defined:

```
enum Status{open, progress, done = 9};     // unscoped enum

auto s1 = open;                            // OK
auto s2 = Status::open;                    // OK
```

```
using enum Status;                        // OK, but no effect

auto s3 = open;                           // OK
auto s4 = Status::open;                   // OK
```

21.3 Delegating Enumeration Types to Different Scopes

Using enum declarations can also be used to delegate enumeration values to different scopes. For example:

```
namespace MyProject {
  class Task {
   public:
    enum class Status{open, progress, done = 9};
    Task();
    ...
  };

  using enum Task::Status;      // expose the values of Status to MyProject
}

auto x = MyProject::open;       // OK: x has value MyProject::Task::open
auto y = MyProject::done;       // OK: y has value MyProject::Task::done
```

Please note that the using enum declaration exposes only the values, not the type:

```
MyProject::Status s;            // ERROR
```

To expose both the type and its values, you also need an ordinary using declaration (type alias):

```
namespace MyProject {
  using Status = Task::Status;  // expose the type Task::Status
  using enum Task::Status;      // expose the values of Task::Status
}

MyProject::Status s = MyProject::done;  // OK
```

For exposed enum values, even argument-dependent lookup (ADL) works fine. For example, we can extend the example above as follows:

```
namespace MyProject {
  void foo(MyProject::Task::Status) {
  }
}

namespace MyScope {
  using enum MyProject::Task::Status;   // OK
}

foo(MyProject::done);  // OK: calls MyProject::foo() with MyProject::Task::Status::done
foo(MyScope::done);     // OK: calls MyProject::foo() with MyProject::Task::Status::done
```

Note that a type alias is not generally used by ADL:

```
namespace MyScope {
  void bar(MyProject::Task::Status) {
  }
  using MyProject::Task::Status;         // expose enum type to MyScope
  using enum MyProject::Task::Status;    // expose the enum values to MyScope
}

MyScope::Status s = MyScope::open;       // OK
bar(MyScope::done);                      // ERROR
MyScope::bar(MyScope::done);             // OK
```

21.4 New Character Type **char8_t**

For better UTF-8 support, C++20 introduces the new character type **char8_t** and a new corresponding string type std::**u8string**.

Type char8_t is a new keyword. It serves as ***the*** type for holding UTF-8 characters and character sequences. For example:

```
char8_t c = u8'@';                  // character with UTF-8 encoding for character @
const char8_t* s = u8"K\u00F6ln";    // character sequence with UTF-8 encoding for Köln
```

The introduction of this type is a breaking change:

- u8 character literals now use char8_t instead of char
- For UTF-8 strings, the new types std::u8string and std::u8string_view are used now

Consider the following example:

```
auto c = u8'@';                   // character with UTF-8 encoding for @
auto s1 = u8"K\u00F6ln";          // character sequence with UTF-8 encoding for Köln
using namespace std::literals;
auto s2 = u8"K\u00F6ln"s;         // string with UTF-8 encoding for Köln
auto sv = u8"K\u00F6ln"sv;        // string view with UTF-8 encoding for Köln
```

The types of c and s have changed here:

- **Before C++20**, this was equivalent to:

```
char c = u8'@';                          // UTF-8 character type before C++20
const char* s1 = u8"K\u00F6ln";          // UTF-8 character sequence type before C++20
using namespace std::literals;
std::string s2 = u8"K\u00F6ln"s;         // UTF-8 string type before C++20
std::string_view sv = u8"K\u00F6ln"s;    // UTF-8 string view type before C++20
```

- **Since C++20**, this is equivalent to:

```
char8_t c = u8'@';                       // UTF-8 character type since C++20
const char8_t* s1 = u8"K\u00F6ln";       // UTF-8 character sequence type since C++20
```

```
using namespace std::literals;
std::u8string s2 = u8"K\u00F6ln"s;          // UTF-8 string type since C++20
std::u8string_view sv = u8"K\u00F6ln"sv; // UTF-8 string view type since C++20
```

The reason for this change is simple: we can now implement special behavior for UTF-8 characters and strings:

- We can overload for UTF-8 character sequences:

  ```
  void store(const char* s)
  {
    storeInFile(convertToUTF8(s));    // store after converting to UTF-8 encoding
  }

  void store(const char8_t* s)
  {
    storeInFile(s);                   // store as is because it already has UTF-8 encoding
  }
  ```

- We can implement special behavior in generic code:

  ```
  void store(const CharT* s)
  {
    if constexpr(std::same_as<CharT, char8_t>) {
      storeInFile(s);                   // store as is because it already has UTF-8 encoding
    }
    else {
      storeInFile(convertToUTF8(s));    // store after converting to UTF-8 encoding
    }
  }
  ```

You can still only store UTF-8 characters of one byte in an object of type `char8_t` (remember that UTF-8 characters have a variable width):

- The character for the at sign, `@`, has the decimal value 64 (hexadecimal `0x40`). Therefore, you can store its value in a `char8_t` and for this reason, the `u8` character literal is well-defined:

  ```
  char8_t c = u8'@';                // OK (c has value 64)
  ```

- The character for the European currency *Euro*, €, consists of three code units: 226 130 172 (hexadecimal: `0xE2 0x82 0xAC`). Therefore, you ***cannot*** store its value in a `char8_t`:

  ```
  char8_t cEuro = u8'€';            // ERROR: invalid character literal
  ```

  Instead, you have to initialize a character sequence or UTF-8 string:

  ```
  const char8_t* cEuro = u8"\u20AC";     // OK
  std::u8string  sEuro = u8"\u20AC";     // OK
  ```

  Here, we use the Unicode notation to specify the value of the UTF-8 character, which creates an array of four `const char8_t` (including the trailing null character), which is then used to initialize `cEuro` and `sEuro`.

If your compiler accepts the character € in your source code (which means it has to support a source file encoding with a character set such as UTF-8 or ISO-8859-15), you could even use this symbol in literals directly:[2]

```
const char8_t* cEuro = u8"€";          // OK if valid character for the compiler
std::u8string  sEuro = u8"€";          // OK if valid character for the compiler
```

However, because this source code is not portable, you should use Unicode characters (such as \u20AC for the € symbol).

Note that a few things in the C++ standard library have changed accordingly:

- Overloads for the new character type char8_t were added
- Functions that use or return UTF-8 strings use the new UTF-8 string types

In addition, note that char8_t is not guaranteed to have 8 bits. It is defined as using an unsigned char internally, which typically has 8 bits, but may have more. As usual, you can use std::numeric_limits<> to check for the number of bits:

```
std::cout << "char8_t has "
          << std::numeric_limits<char8_t>::digits << " bits\n";
```

21.4.1 Changes in the C++ Standard Library for char8_t

The C++ standard library changed the following to support char8_t:

- u8string is now provided (defined as std::basic_string<char8_t>)
- u8string_view is now provided (defined as std::basic_string_view<char8_t>)
- std::numeric_limits<char8_t> is now defined
- std::char_traits<char8_t> is now defined
- Hash functions for char8_t strings and string views are now provided
- std::mbrtoc8() and c8rtomb() are now provided
- codecvt facets for conversions between char8_t and char16_t or char32_t are now provided
- For filesystem paths, u8string() now returns std::u8string instead of std::string
- std::atomic<char8_t> is now provided

Note that these changes might break existing code when it is being compiled with C++20, which is discussed next.

21.4.2 Broken Backward Compatibility

Because C++20 changes the type of UTF-8 literals and signatures that return UTF-8 strings, code that uses UTF-8 characters might no longer compile.

[2] Thanks to Tom Honermann for pointing this out.

Broken Code Using the Character Type

For example:

```
std::string s0 = u8"text";   // OK in C++17, ERROR since C++20

auto s = u8"K\u00F6ln";      // s is const char* until C++17, but const char8_t* since C++20
const char* s2 = s;          // OK in C++17, ERROR since C++20
std::cout << s << '\n';      // OK in C++17, ERROR since C++20

auto c = u8'c';              // c1 is char in C++17, but char8_t since C++20
char c2 = c;                 // OK (even if char8_t)
char* cp = &c;               // OK in C++17, ERROR since C++20
std::cout << c;              // OK in C++17, ERROR since C++20
```

Broken Code Using the New String Types

In particular, code that returns UTF-8 strings might now lead to problems.

For example, the following code no longer compiles because `u8string()` now returns a `std::u8string` instead of a `std::string`:

```
// iterate over directory entries:
for (const auto& entry : fs::directory_iterator(path)) {
  std::string name = entry.path().u8string();     // OK in C++17, ERROR since C++20
  ...
}
```

You have to adjust the code by using a different type, `auto`, or support both types:

```
// iterate over directory entries (C++17 and C++20):
for (const auto& entry : fs::directory_iterator(path)) {
#ifdef __cpp_char8_t
  std::u8string name = entry.path().u8string();   // OK since C++20
#else
  std::string name = entry.path().u8string();     // OK in C++17
#endif
  ...
}
```

Broken Code I/O for UTF-8 Strings

You can also no longer print UTF-8 characters or strings to `std::cout` (or any other standard output stream):

```
std::cout << u8"text";        // OK in C++17, ERROR since C++20
std::cout << u8'X';           // OK in C++17, ERROR since C++20
```

In fact, C++20 deleted the output operator for all extended character types (`wchar_t`, `char8_t`, `char16_t`, `char32_t`) unless the output stream supports the same character type:

```
std::cout << "text";             // OK
std::cout << L"text";            // wchar_t string: OOPS in C++17, ERROR since C++20
std::cout << u8"text";           // UTF-8 string: OK in C++17, ERROR since C++20
std::cout << u"text";            // UTF-16 string: OOPS in C++17, ERROR since C++20
std::cout << U"text";            // UTF-32 string: OOPS in C++17, ERROR since C++20

std::wcout << "text";            // OK
std::wcout << L"text";           // OK
std::wcout << u8"text";          // UTF-8 string: OK in C++17, ERROR since C++20
std::wcout << u"text";           // UTF-16 string: OOPS in C++17, ERROR since C++20
std::wcout << U"text";           // UTF-32 string: OOPS in C++17, ERROR since C++20
```

Note the statements marked with *OOPS*: they all compiled in C++17, but they printed the addresses of the strings instead of their values. Therefore, C++20 disables not only the output of UTF-8 characters, but also output that did not work at all.

Dealing with Broken Code

You now be wondering how to deal with code that formerly worked with UTF-8 characters. The easiest way to deal with it is to use reinterpret_cast<>:

```
auto s = u8"text";                              // s is const char8_t* since C++20
std::cout << s;                                 // ERROR since C++20
std::cout << reinterpret_cast<const char*>(s);  // OK
```

For a single character, using a static_cast is enough:

```
auto c = u8'x';                                 // c is char8_t since C++20
std::cout << c;                                 // ERROR since C++20
std::cout << static_cast<char>(c);              // OK
```

You can bind its use or the use of other workarounds on the feature test macro for the char8_t characters feature:[3]

```
auto s = u8"text";                              // s is const char8_t* since C++20
#ifdef __cpp_char8_t
std::cout << reinterpret_cast<const char *>(s); // OK
#else
std::cout << s;                                 // OK in C++17, ERROR since C++20
#endif
```

If you are wondering why C++20 does not provide a working output operator for UTF-8 characters, please note that this is a pretty complex issue that we ha simply not enough time to solve yet. You can read more about this here: http://stackoverflow.com/a/58895428.

Because using reinterpret_cast<> might not scale when UTF-8 characters are used heavily, Tom Honermann wrote a guideline on how to deal with code for UTF-8 characters before and since C++20: "*P1423R3* char8_t *Backward Compatibility Remediation*." If you deal with UTF-8 characters and strings, you should definitely read it. You can download it from http://wg21.link/p1423.

[3] For the final behavior specified with C++20, __cpp_char8_t should have (at least) the value 201907.

21.5 Improvements for Aggregates

C++20 provides a couple of improvements for aggregates, which are described in this section:

- Designated initializers (initial values for specific members) are (partially) supported
- You can initialize aggregates with parentheses
- The fixed definition of aggregates and the consequence for `std::is_default_constructible<>`

In addition, other sections of this book describe additional new aspects of aggregates when they are used in generic code:

- Using class template argument deduction (CTAD) for aggregates
- Aggregates can be used as non-type template parameters (NTTP)

21.5.1 Designated Initializers

For aggregates, C++20 provides a way to specify which member should be initialized with a passed initial value. However, you can use this only to skip parameters.

For example, assume we have the following aggregate type:

```
struct Value {
  double amount = 0;
  int precision = 2;
  std::string unit = "Dollar";
};
```

Then, the following way to initialize values of this type is supported now:

```
Value v1{100};                           // OK (not designated initializers)
Value v2{.amount = 100, .unit = "Euro"}; // OK (second member has default value)
Value v3{.precision = 8, .unit = "$"};   // OK (first member has default value)
```

See *lang/designated.cpp* for a complete example.

Note the following constraints:

- You have to pass the initial value with = or `{}`.
- You can skip members, but you have to follow their order. The order of members that are initialized by name must match their order in the declaration.
- You must use designated initializers for either all or none of the arguments. Mixed initialization is not allowed.
- Using designated initialization for arrays is not supported.
- Nested initialization with designated initializers is possible, but not directly using `.mem.mem`.
- You cannot use designated initializers when initializing aggregates with parentheses.
- Designated initializers can also be used in unions.

For example:

```
Value v4{100, .unit = "Euro"};           // ERROR: all or none designated
Value v5{.unit = "$", .amount = 20};     // ERROR: invalid order
Value v6(.amount = 29.9, .unit = "Euro"); // ERROR: only supported for curly braces
```

The restrictions that designated initializers follow the order of members, can be used either for all or for none of the arguments, do not support direct nesting, and do not support arrays are restrictions compared to the programming language C. The reason for respecting the order of the members is to ensure that the initialization reflects the order in which constructors are called (which is the opposite order destructors are called).

And as an example of nested initialization with = and {} and unions:

```
union Sub {
  double x = 0;
  int y = 0;
};

struct Data {
  std::string name;
  Sub val;
};

Data d1{.val{.y=42}};         // OK
Data d2{.val = {.y{42}}};     // OK
```

You cannot directly nest designated initializers:

```
Data d2{.val.y = 42};         // ERROR
```

21.5.2 Aggregate Initialization with Parentheses

Assume you have declared the following aggregate:

```
struct Aggr {
  std::string msg;
  int val;
};
```

Before C++20, you could only use curly braces to initialize aggregates with values:

```
Aggr a0;                    // OK, but no initialization
Aggr a1{};                  // OK, value initialized with "" and 0
Aggr a2{"hi"};              // OK, initialized with "hi" and 0
Aggr a3{"hi", 42};          // OK, initialized with "hi" and 42
Aggr a4 = {};               // OK, initialized with "" and 0
Aggr a5 = {"hi"};           // OK, initialized with "hi" and 0
Aggr a6 = {"hi", 42};       // OK, initialized with "hi" and 42
```

Since C++20, you can also use parentheses as outer characters for a direct initialization without =:

```
Aggr a7("hi");              // OK since C++20: initialized with "hi" and 0
Aggr a8("hi", 42);          // OK since C++20: initialized with "hi" and 42
Aggr a9({"hi", 42});        // OK since C++20: initialized with "hi" and 42
```

Using = or inner parentheses still does not work:

```
Aggr a10 = "hi";           // ERROR
Aggr a11 = ("hi", 42);     // ERROR
Aggr a12(("hi", 42));      // ERROR
```

Using inner parentheses might even compile. In that case, they are used as any parentheses around an expression that uses the comma operator.

Note that you can use parentheses even to initialize an array of unknown bounds:

```
int a1[]{1, 2, 3};         // OK since C++11
int a2[](1, 2, 3);         // OK since C++20
int a3[] = {1, 2, 3};      // OK
int a4[] = (1, 2, 3);      // still ERROR
```

However, “brace elision” is not supported (there are no nested braces to elide):

```
struct Arr {
  int elem[10];
};

Arr arr1{1, 2, 3};      // OK
Arr arr2(1, 2, 3);      // ERROR
Arr arr3{{1, 2, 3}};    // OK
Arr arr4({1, 2, 3});    // OK (even before C++20)
```

Therefore, to initialize `std::array`s, you still have to use braces:

```
std::array<int,3> a1{1, 2, 3};   // OK: shortcut for std::array{{1, 2, 3}}
std::array<int,3> a2(1, 2, 3);   // still ERROR
```

Reason for Aggregate Initialization with Parentheses

The reason for supporting aggregate initialization with parentheses is that this allows you to call the operator `new` with parentheses:

```
struct Aggr {
  std::string msg;
  int val;
};

auto p1 = new Aggr{"Rome", 200};   // OK since C++11
auto p2 = new Aggr("Rome", 200);   // OK since C++20 (ERROR before C++20)
```

This helps to support the use of aggregates in types that call `new` internally with parentheses to store values in existing memory as is the case for containers and smart pointers. In fact, the following is possible since C++20:

- You can now use `std::make_unique<>()` and `std::make_shared<>()` for aggregates:

  ```
  auto up = std::make_unique<Aggr>("Rome", 200);   // OK since C++20
  auto sp = std::make_shared<Aggr>("Rome", 200);   // OK since C++20
  ```

  There was no way to use these helper functions for aggregates before C++20.

- You can now emplace new values into containers of aggregates:

  ```
  std::vector<Aggr> cont;
  cont.emplace_back("Rome", 200);                // OK since C++20
  ```

Note that there are still types that cannot be initialized with parentheses but that can be initialized with curly braces: scoped enumerations (`enum class` types). Type `std::byte` (introduced with C++17) is one example:

```
std::byte b1{0};                                      // OK
std::byte b2(0);                                      // still ERROR
auto upb2 = std::make_unique<std::byte>(0);           // still ERROR
auto upb3 = std::make_unique<std::byte>(std::byte{0}); // OK
```

For `std::array`, you still need braces (as stated above):

```
std::vector<std::array<int, 3>> ca;
ca.emplace_back(1, 2, 3);           // ERROR
ca.emplace_back({1, 2, 3});         // ERROR
ca.push_back({1, 2, 3});            // still OK
```

Aggregate Initialization with Parentheses in Detail

The proposal to introduce initialization with parentheses lists the following design guidelines:

- Any existing meaning of `Type(val)` should not change.
- Parenthesized initialization and braced initialization should be as similar as possible but also as distinct as necessary to conform with the existing mental models of braced lists and parenthesized lists.

And in fact, there are the following differences between aggregate initialization with curly braces and aggregate initialization with parentheses:

- Initialization with parentheses does not detect narrowing conversions.
- Initialization with parentheses allows all implicit conversions (not only from derived to base class).
- When using parentheses, reference members do not extend the lifetime of passed temporary objects.
- Using parentheses does not support brace elisions (using them is like passing arguments to parameters).
- An initialization with empty parentheses works even for `explicit` members.
- Using parentheses does not support designated initializers.

An example of the lack of detecting narrowing is the following:

```
struct Aggr {
  std::string msg;
  int val;
};

Aggr a1{"hi", 1.9};                 // ERROR: narrowing
Aggr a2("hi", 1.9);                 // OK, but initializes with 1

std::vector<Aggr> cont;
cont.emplace_back("Rome", 1.9);     // initializes with 1
```

Note that emplace functions never detect narrowing.

An example of the difference when dealing with implicit conversions is this:

```
struct Other {
  ...
  operator Aggr();             // defines implicit conversion to Aggr
};

Other o;
Aggr a7{o};                    // ERROR: no implicit conversion supported
Aggr a8(o);                    // OK, implicit conversion possible
```

Note that the conversions cannot be defined by the aggregate itself because aggregates cannot have a user-defined constructor.

The missing brace elision and the complex rules of brace initialization lead to the following behavior:

```
Aggr a01{"x", 65};             // init string with "x" and int with 65
Aggr a02("x", 65);             // OK since C++20 (same effect)

Aggr a11{{"x", 65}};           // runtime ERROR: "x" doesn't have 65 characters
Aggr a12({"x", 65});           // OK even before C++20: init string with "x" and int with 65

Aggr a21{{{"x", 65}}};         // ERROR: cannot initialize string with initializer list of "x" and 65
Aggr a22({{"x", 65}});         // runtime ERROR: "x" doesn't have 65 characters

Aggr a31{'x', 65};             // ERROR: cannot initialize string with 'x'
Aggr a32('x', 65);             // ERROR: cannot initialize string with 'x'

Aggr a41{{'x', 65}};           // init string with 'x' and char(65)
Aggr a42({'x', 65});           // OK since C++20 (same effect)

Aggr a51{{{'x', 65}}};         // init string with 'x' and char(65)
Aggr a52({{'x', 65}});         // OK even before C++20: init string with 'x' and 65

Aggr a61{{{{'x', 65}}}};       // ERROR
Aggr a62({{{'x', 65}}});       // OK even before C++20: init string with 'x' and 65
```

When performing *copy initialization* (initializing with =), `explicit` matters and might have a different effect with empty parentheses as follows:[4]

```
struct C {
  explicit C() = default;
};

struct A {    // aggregate
  int i;
  C c;
};
```

[4] Thanks to Timur Doumler for pointing this out.

```
auto a1 = A{42, C{}};          // OK: explicit initialization
auto a2 = A(42, C());          // OK since C++20: explicit initialization

auto a3 = A{42};               // ERROR: cannot call explicit constructor
auto a4 = A(42);               // ERROR: cannot call explicit constructor

auto a5 = A{};                 // ERROR: cannot call explicit constructor
auto a6 = A();                 // OK: can call explicit constructor
```

However, this is nothing new in C++20 because the initialization of `a6` was already supported before C++20.

Finally, if you use parentheses for the initialization of an aggregate that has rvalue reference members, the lifetime of initial values is not extended. Thus, you should not pass temporary objects (prvalues):[5]

```
struct A {
  int a;
  int&& r;
};

int f();
int n = 10;

A a1{1, f()};                  // OK, lifetime is extended
std::cout << a1.r << '\n';     // OK
A a2(1, f());                  // OOPS: dangling reference
std::cout << a2.r << '\n';     // runtime ERROR
A a3(1, std::move(n));         // OK as long as a3 is used only while n exists
std::cout << a3.r << '\n';     // OK
```

Due to these complicated rules and traps, you should use aggregate initialization with parentheses only when you have to, such as when using `std::make_unique<>()`, `std::make_shared<>()` or emplace functions.

21.5.3 Definition of Aggregates

Once again, the definition of *aggregate* changed with C++20. This time, the modification reversed an extension that came in with C++11 and turned out to be an error. From C++11 until C++17, *user-**provided*** constructors were not allowed. For this reason, the following was a legal definition of an aggregate:

```
struct A {             // aggregate from C++11 until C++17
  ...
  A() = delete;        // user-declared, but not user-provided constructor
};
```

[5] Thanks to Ville Voutilainen for pointing this out.

With this definition, the aggregate initialization was valid even though the type had a deleted default constructor:

```
A a1;                   // ERROR
A a2{};                 // OK from C++11 until C++17
```

This did not apply only to default constructors. For example:

```
struct D {              // aggregate from C++11 until C++17
  int i = 0;
  D(int) = delete;      // user-declared, but not user-provided constructor
};

D d1(3);                // ERROR
D d2{3};                // OK from C++11 until C++17
```

This special behavior was introduced for a very special case but was totally counter-intuitive.[6] C++20 fixed this by going back to requiring for aggregates that there is no *user-**declared*** constructor (as was the case before C++11):

```
struct A {          // NO aggregate since C++20
  ...
  A() = delete;     // user-declared, but not user-provided constructor
};

A a1;         // ERROR
A a2{};       // ERROR since C++20
```

For this reason, the type trait `std::is_default_constructible_v<>` will no longer be `true` for type `A` and `D` as declared above.

However, note that some programmers did use this feature to force (possibly empty) curly braces when creating an object of an aggregate type (which does ensure that members are always initialized). For them, the workaround to get the same behavior is to derive from a base type as follows:

```
struct MustInit {
  MustInit(MustInit&&) = default;
};

struct A : MustInit {
  ...
};

A a1;         // ERROR
A a2{};       // OK
```

[6] The feature was introduced as a “hack” to support C compatibility when initializing atomic types. However, even several committee members were not aware of that behavior and were very surprised and it even turned out that this support was never needed.

To summarize, since C++20, an *aggregate* is defined as follows:

- Either an array
- Or a *class type* (`class`, `struct`, or `union`) with:
  - No *user-**declared*** constructor
  - No constructor inherited by a `using` declaration
  - No `private` or `protected` non-static data members
  - No `virtual` functions
  - No `virtual`, `private`, or `protected` base class

To allow you to *initialize* aggregates, the following additional constraints apply:

- No `private` or `protected` base class members
- No `private` or `protected` constructors

21.6 New Attributes and Attribute Features

Since C++11, it has been possible to specify *attributes* (formal annotations that enable or disable warnings). With C++20, again new attributes were introduced and existing attributes were extended.

21.6.1 Attributes `[[likely]]` and `[[unlikely]]`

The new attributes `[[likely]]` and `[[unlikely]]` were introduced by C++20 to help the compiler when performing branch optimizations.

When there are multiple paths in the code, you can use these attributes to give the compiler a hint about which path is most likely or not likely.

For example:

```
int f(int n)
{
  if (n <= 0) [[unlikely]] {   // n <= 0 is considered to be arbitrarily unlikely
    return n;
  }
  else {
    return n * n;
  }
}
```

This might, for example, force the compiler to generate assembler code that directly handles the `else` branch while jumping to later assembler commands for the `then` case.

It probably has but is not guaranteed to have the same effect as:

```
int f(int n)
{
  if (n <= 0) {
    return n;
  }
  else [[likely]] {             // n > 0 is considered to be arbitrarily likely
    return n * n;
  }
}
```

Here is another example:

```
int g(int n)
{
  switch (n) {
    case 1:
      ...
      break;
    [[likely]] case 2:          // n == 2 is considered to be arbitrarily most likely
      ...
      break;
  }
  ...
}
```

The effect of these attributes is compiler-specific and there is no guarantee that the attributes have any impact at all.

You should be careful when using these attributes and double-check their effect. Often, compilers know better how to optimize code, meaning that excessive usage of these attributes might be counter-productive.

21.6.2 Attribute `[[no_unique_address]]`

Classes often have members that impact the behavior but do not provide a state. Examples are the hash function of an unordered container, the deleter of a `std::unique_ptr`, or the standard allocator of containers or strings. All they provide are member functions (and static members) but no non-static data members.

However, members usually need memory even though they do not store anything. For example, consider the following code:

```
struct Empty {};     // empty class: size is usually 1

struct I {           // size is same as sizeof(int)
  int i;
};

struct EandI {       // size is sum of size of members with alignment
  Empty e;
  int i;
};

std::cout << "sizeof(Empty):   " << sizeof(Empty) << '\n';
std::cout << "sizeof(I):       " << sizeof(I) << '\n'
std::cout << "sizeof(EandI):   " << sizeof(EandI) << '\n';
```

Depending on the size of `int`, the output might look something like this:

```
sizeof(E):      1
sizeof(I):      4
sizeof(EandI):  8
```

This is an unnecessary waste of space. Before C++20, you could use the *empty base class optimization (EBCO)* to avoid the unnecessary overhead. By deriving from a class with no data members, compilers are allowed to save the corresponding space:

```
struct EbasedI : Empty {      // using EBCO
  int i;
};

std::cout << "sizeof(EbasedI): " << sizeof(EbasedI) << '\n';
```

On a platform with the sizes above, this outputs:

```
sizeof(EbasedI): 4
```

However, this workaround is a bit clumsy and might not always work. For example, you cannot use EBCO if the empty base class is `final`.

Since C++20, there is a different way to get the same effect. You only have to declare the members that provide no state with the attribute `[[no_unique_address]]`:

```
struct EattrI {               // same effect as EBCO
  [[no_unique_address]] Empty e;
  int i;
};

struct IattrE {               // same effect as EBCO
  int i;
  [[no_unique_address]] Empty e;
};

std::cout << "sizeof(EattrI):  " << sizeof(EattrI) << '\n';
std::cout << "sizeof(IattrE):  " << sizeof(IattrE) << '\n';
```

On the platform above where this attribute is supported, the output becomes:

```
sizeof(EattrI):  4
sizeof(IattrE):  4
```

Note that compilers are not required to honor this attribute.

The member marked with `[[no_unique_address]]` still counts as a member for initialization:

```
EattrI ei = {42};             // ERROR: can't initialize member e with 42
EattrI ei = {{},42};          // OK
```

This optimization also means that the address of the member `e` gets the same address as the member `i` of the same object. This still means that the members `e` of two different objects have different addresses.

If a data type only has data members with this attribute, it is implementation-defined whether the type trait `std::is_empty_v<>` yields `true`:

```
struct OnlyEmpty {
  [[no_unique_address]] Empty e;
};

std::is_empty_v<OnlyEmpty>    // might yield true or false
```

Finally, note that Visual C++ currently ignores this attribute. The reason is that Visual C++ initially allowed this attribute without honoring it and now honoring it becomes an ABI break. Visual C++ might support it in a future ABI-breaking version, though. Until then, you can use `[[msvc::no_unique_address]]` instead:

```
struct EattrI {             // also works for Visual C++
    [[no_unique_address]] [[msvc::no_unique_address]] Empty e;
    Type i;
};
```

21.6.3 Attribute `[[nodiscard]]` with Parameter

C++17 introduced the attribute `[[nodiscard]]`, which can be used to encourage warnings by the compiler if a return value of a function is not used.

`[[nodiscard]]` should usually be used to signal misbehavior when return values are not used. The misbehavior might be:

- **Memory leaks**, such as not using returned allocated memory
- **Unexpected or non-intuitive behavior**, such as getting different/unexpected behavior when not using the return value
- **Unnecessary overhead**, such as calling something that is a no-op if the return value is not used

However, there was no way to specify a message for a reason to explain why a programmer should use the return values. C++20 introduced an optional parameter for that.

For example:

```
class MyType {
  public:
    ...
    [[nodiscard("Possible memory leak")]]        // OK, since C++20
    char* release();

    void clear();

    [[nodiscard("Did you mean clear()?")]]     // OK, since C++20
    bool empty() const;
};
```

The second declaration requests that the compiler prints the warning message “`Did you mean clear()?`” when the return value of `empty()` is not used. In fact, C++20 introduced this attribute for the member function `empty()` of all standard containers and according to feedback, we know that compilers did indeed find errors where programmers thought they had requested making the collection empty.

21.7 Feature Test Macros

Each and every C++ version introduces various language and library features that are supported by compilers step by step. For that reason, knowing which C++ version a compiler supports in general is often not enough; for portable code, it might be important to know whether a specific feature is available.

For that purpose, C++20 officially introduces ***feature test macros***. For each new language and library feature, a macro can be used to signal whether the feature is available. The macro can even provide information about which version of a feature is supported.

For example, the following source code will use different code depending on whether (and in which form) generic lambdas are available:

```
#ifdef __cpp_generic_lambdas
#if __cpp_generic_lambdas >= 201707
...     // generic lambdas with template parameters can be used
#else
...     // generic lambdas can be used
#else
...     // no generic lambdas can be used
#endif
```

All feature test macros for language features start with `__cpp`.

As another example, the following code provides and uses a workaround if `std::as_const()` is not available yet:

```
#ifndef __cpp_lib_as_const
template<typename T>
const T& asConst(T& t) {
  return t;
}
#endif

#ifdef __cpp_lib_as_const
  auto printColl = [&coll = std::as_const(coll)] {
#else
  auto printColl = [&coll = asConst(coll)] {
#endif
    ...
  };
```

All feature test macros for library features start with `__cpp_lib`.

Feature test macros for language features are defined by the compilers. Feature test macros for library features are provided by the new header `<version>`.

See the use of `__cpp_char8_t` as another example of making code that uses UTF-8 characters portable before and after C++20.

21.8 Afternotes

The range-based `for` loop with initialization was first proposed by Thomas Köppe in http://wg21.link/p0614r0. The finally accepted wording was formulated by Thomas Köppe in http://wg21.link/p0614r1.

`using` for enumeration values was first proposed by Gasper Azman and Jonathan Müller in http://wg21.link/p1099r0. The finally accepted wording was formulated by Gasper Azman and Jonathan Müller in http://wg21.link/p1099r5.

The request for a distinct type for UTF-8 characters was first proposed by Tom Honermann in http://wg21.link/p0482r0. The finally accepted wording was formulated by Tom Honermann in http://wg21.link/p0482r6. The corresponding fixes to the output operators were accepted as proposed by Tom Honermann in http://wg21.link/p1423r3.

The request to support designated initializers was first proposed by Tim Shen, Richard Smith, Zhihao Yuan, and Chandler Carruth in http://wg21.link/p0329r0. The finally accepted wording was formulated by Tim Shen and Richard Smith in http://wg21.link/p0329r4.

Aggregate initialization with parentheses was first proposed by Ville Voutilainen in http://wg21.link/p0960r0. The finally accepted wording was formulated by Ville Voutilainen and Thomas Köppe in http://wg21.link/p0960r3.

The change of the definition of an aggregate was accepted as proposed by Timur Doumler, Arthur O'Dwyer, Richard Smith, Howard E. Hinnant, and Nicolai Josuttis in http://wg21.link/p1008r1.

The attributes `[[likely]]` and `[[unlikely]]` were accepted as proposed by Clay Trychta in http://wg21.link/p0479r5.

The attribute `[[no_unique_address]]` was accepted as proposed by Richard Smith in http://wg21.link/p0840r2.

Allowing the attribute `[[nodiscard]]` to have an argument was accepted as proposed by JeanHeyd Meneide and Isabella Muerte in http://wg21.link/p1301r4.

Feature test macros were adopted into C++ as proposed by Ville Voutilainen and Jonathan Wakely in http://wg21.link/p0941r2.

Chapter 22

Small Improvements for Generic Programming

This chapter presents additional features and extensions of C++20 for generic programming that have not been covered in this book yet.

22.1 Implicit `typename` for Type Members of Template Parameters

When using type members of template parameters, you usually have to qualify this use with the keyword `typename`:

```
template<typename T>
typename T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = typename T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

Before C++20, all qualifications of the type members `value_type` and `iterator` of `T` were necessary. Since C++20, you can skip `typename` in contexts where it is clear that a type is passed. In this case, this applies to the specification of the return type and the type used in the alias declaration (where `using` introduces a new name for a type):

```
template<typename T>
T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

Note that the parameter `pos` and the variable `elem` still need `typename`. The most important places where you can skip `typename` now are:

- When declaring return types (except for a local forward declaration)
- When declaring members in class templates
- When declaring parameters of member or `friend` functions in class templates
- When declaring parameters of `requires` expressions
- For the types in alias declarations

In particular, when declaring a class template, you can now skip `typename` in most cases:

```
template<typename T>
class MyClass {
  T::value_type val;                // no need for typename since C++20
 public:
  ...
  T::iterator begin() const;        // no need for typename since C++20
  T::iterator end() const;          // no need for typename since C++20
  void print(T::iterator) const;    // no need for typename since C++20
};
```

However, because the rules for implicit `typename` are pretty subtle to some extent, you might simply still always use `typename` when using the type member of a template parameter (at least outside class templates).[1]

22.1.1 Rules for Implicit `typename` in Detail

Since C++20, you can skip `typename` when using a type member for a template parameter in the following situations:

- In an alias declaration (i.e., when declaring a type name with `using`); note that a type declaration with `typedef` still needs `typename`
- When defining or declaring the return type of a function (unless the declaration happens inside a function or block scope)
- When declaring a trailing return type
- When specifying the target type for `static_cast`, `const_cast`, `reinterpret_cast`, or `dynamic_cast`
- When specifying the type for `new`
- Inside a class when
  - Declaring a data member
  - Declaring the return type of a member function
  - Declaring a parameter of a member or friend function or lambda (a default argument might still need it)
- When declaring parameter types in a `requires` expression
- When declaring a default value for a type parameter of a template
- When declaring the type of a non-type template parameter

[1] Thanks to Arthur O'Dwyer for pointing this out.

Note that before C++20, `typename` was not necessary in a few other situations:

- When specifying the base type of an inherited class
- When passing initial values to the base class in a constructor
- When using a type member inside the class declaration

The following example demonstrates most of the cases above (here, `TYPENAME` is used where `typename` is optional since C++20):

```
template<typename T,
         auto ValT = typename T::value_type{}>     // typename required
class MyClass {
  TYPENAME T::value_type val;                      // typename optional
 public:
  using iterator = TYPENAME T::iterator;           // typename optional

  TYPENAME T::iterator begin() const;              // typename optional
  TYPENAME T::iterator end() const;                // typename optional
  void print(TYPENAME T::iterator) const;          // typename optional
  template<typename T2 = TYPENAME T::value_type>   // second typename optional
    void assign(T2);
};

template<typename T>
TYPENAME T::value_type                             // typename optional
foo(const T& cont, typename T::value_type arg)     // typename required
{
  typedef typename T::value_type ValT2;            // typename required
  using ValT1 = TYPENAME T::value_type;            // typename optional
  typename T::value_type val;                      // typename required

  typename T::value_type other1(void);             // typename required
  auto other2(void) -> TYPENAME T::value_type;     // typename optional

  auto l1 = [] (TYPENAME T::value_type) {          // typename optional
            };

  auto p = new TYPENAME T::value_type;             // typename optional
  val = static_cast<TYPENAME T::value_type>(0);    // typename optional
  ...
}
```

22.2 Improvements for Aggregates in Generic Code

C++20 provides a couple of improvements for aggregates. For generic code we now have:

- Using class template argument deduction (CTAD) for aggregates
- Aggregates can be used as non-type template parameters (NTTP)

The former is described in this section.

Note that there are other new features for aggregates:

- Designated initializers (initial values for specific members) are (partially) supported
- You can initialize aggregates with parentheses
- The fixed definition of aggregates and the consequence for `std::is_default_constructible<>`

22.2.1 Class Template Argument Deduction (CTAD) for Aggregates

Since C++17, constructors can be used to deduce template parameters of class templates. For example:

```
template<typename T>
class Type {
  T value;
 public:
  Type(T val)
   : value{val} {
  }
  ...
};

Type<int> t1{42};
Type t2{42};        // deduces Type<int> since C++17
```

However, even for simple aggregates, a similar deduction according to the way objects are initialized was not supported:

```
template<typename T>
struct Aggr {
  T value;
};

Aggr<int> a1{42};   // OK
Aggr a2{42};        // ERROR before C++20
```

You had to provide a deduction guide:

```
template<typename T>
struct Aggr {
  T value;
};
```

```
template<typename T>
Aggr(T) -> Aggr<T>;

Aggr<int> a1{42};   // OK
Aggr a2{42};        // OK since C++17
```

Since C++20, there is no longer any need for a deduction guide, meaning that the following is enough:

```
template<typename T>
struct Aggr {
  T value;
};

Aggr<int> a1{42};   // OK
Aggr a2{42};        // OK since C++20 even without deduction guide
```

Note that this feature also works when using parentheses to initialize aggregates:

```
Aggr a3(42);        // OK since C++20 even without deduction guide
```

The deduction rules can be subtle, as the following example demonstrates:

```
template<typename T>
struct S {
  T x;
  T y;
};

template<typename T>
struct C {
  S<T> s;
  T x;
  T y;
};

C c1 = {{1, 2}, 3, 4};        // OK, C<int> deduced
C c2 = {{1, 2}, 3};           // OK, C<int> deduced (y is 0)
C c3 = {{1, 2}, 3.3, 4.4};    // OK, C<double> deduced

C c4 = {{1, 2}, 3, 4.4};      // ERROR: int and double deduced for T
C c5 = {{1, 2}};              // ERROR: T only indirectly deduced
C c6 = {1, 2, 3};             // ERROR: don't know how many values S<T> needs
C<int> c7 = {1, 2, 3};        // OK
```

Note that class template argument deduction even works for aggregates with a variadic number of elements:

```
// aggregate with variadic number of base types:
template<typename... T>
struct C : T... {
};

struct Base1 {
};
struct Base2 {
};

// aggregate initialized with two elements of types Base1 and Base2:
C c1{Base1{}, Base2{}};

// aggregate initialized with three lambda elements:
C c2{[] { f1(); },
     [] { f2(); },
     [] { f3(); }
    };
```

22.3 Conditional `explicit`

To disable implicit type conversions, constructors can be declared as `explicit`. However, for generic code, you might want to make the constructors `explicit` if and only if a type parameter has explicit constructors. That way, you can perfectly delegate type conversion support to a wrapper type.

The way to specify a conditional `explicit` is like specifying a conditional `noexcept`. Directly after `explicit`, you can specify a Boolean compile-time expression in parentheses. Here is an example:

lang/wrapper.hpp

```
#include <type_traits>   // for std::is_convertible_v<>

template<typename T>
class Wrapper {
    T value;
  public:
    template<typename U>
    explicit(!std::is_convertible_v<U, T>)
    Wrapper(const U& val)
     :value{val} {
    }
    ...
};
```

The class template `Wrapper<>` that holds values of type `T` has a generic constructor, meaning that you could initialize the value with any type that is implicitly convertible to `T`. If there is no implicit conversion from `U` to `T`, the constructor is `explicit`.

As a consequence, you can initialize a `Wrapper` of a type only if there is an implicit type conversion enabled. For example:

lang/explicitwrapper.cpp

```
#include "wrapper.hpp"
#include <string>
#include <vector>

void printStringWrapper(Wrapper<std::string>) {
}
void printVectorWrapper(Wrapper<std::vector<std::string>>) {
}

int main()
{
  // implicit conversion from string literal to string:
  std::string s1{"hello"};
  std::string s2 = "hello";                     // OK
  Wrapper<std::string> ws1{"hello"};
  Wrapper<std::string> ws2 = "hello";           // OK
  printStringWrapper("hello");                  // OK

  // NO implicit conversion from size to vector<string>:
  std::vector<std::string> v1{42u};
  std::vector<std::string> v2 = 42u;            // ERROR: explicit
  Wrapper<std::vector<std::string>> wv1{42u};
  Wrapper<std::vector<std::string>> wv2 = 4u2;  // ERROR: explicit
  printVectorWrapper(42u);                      // ERROR: explicit
}
```

To demonstrate the effect of a conditional `explicit`, we use a `Wrapper<>` for strings and for vectors of strings:

- For type `std::string`, the constructor enables implicit conversions from string literals. Therefore, the constructor of type `Wrapper<>` is not `explicit`, which has the effect that we pass string literals to copy initialize a string wrapper or pass a string literal to a function that takes a string wrapper.[2]
- For type `std::vector<std::string>`, the constructor that takes an unsigned size is declared as `explicit` in the C++ standard library. For that reason, `std::is_convertible<>` from a size to the vector is `false` and the `Wrapper<>` constructor becomes `explicit`. Therefore, we also cannot pass a size to initialize a wrapper of a vector of strings or pass a size to a function that takes that wrapper type.

[2] Remember that implicit conversions are required for copy initialization (initialization with =) and parameter passing.

The conditional `explicit` can be used for any place where `explicit` can be used. Therefore, you could also use it to make conversion operators conditional `explicit`:

```
template<typename T>
class MyType {
  public:
    ...
    explicit(!std::is_convertible_v<T, bool>) operator bool();
};
```

However, I have no useful example of that. Conversions to `bool`, which is by far the most useful application of conversion operators, should always be `explicit` so that you do not accidentally pass a `MyType` object to a function that expects a Boolean value.

22.3.1 Conditional `explicit` in the Standard Library

The C++ standard library uses a conditional `explicit` in a couple of places. For example, `std::pair<>` and `std::tuple<>` use it to support assignments of pairs and tuples of slightly different types only if there are implicit conversions available.

For example:

```
std::pair<int, int> p1{11, 11};

std::pair<long, long> p2{};
p2 = p1;                          // OK: implicit conversion from int to long

std::pair<std::chrono::day, std::chrono::month> p3{};
p3 = p1;                                                    // ERROR
p3 = std::pair<std::chrono::day, std::chrono::month>{p1};   // OK
```

Because the constructors of chrono date types that take an integral value are `explicit`, assignments from a pair of `int` to a pair of day and month fails. You have to use an explicit conversion.

This also applies when inserting element into maps (because the elements are key/value pairs):

```
std::map<std::string, std::string> coll1;
coll1.insert({"hi", "ho"});                 // OK: uses implicit conversions to strings

std::map<std::string, std::chrono::month> coll2;
coll2.insert({"XI", 11});                   // ERROR: no implicit conversion that fits
coll2.insert({"XI", std::chrono::month{11}}); // OK (inserts elem with string and month)
```

This behavior of `std::pair<>` is not new. However, before C++20, implementations of the standard library had to use SFINAE to implement the conditional behavior of `explicit` (declaring two constructors and disabling one of them if the condition is not met).

As another example, the constructors of `std::span<>` are conditional `explicit`:

```
namespace std {
  template<typename ElementType, size_t Extent = dynamic_extent>
  class span {
   public:
    static constexpr size_type extent = Extent;
    ...
    constexpr span() noexcept;
    template<typename It>
      constexpr explicit(extent != dynamic_extent)
      span(It first, size_type count);
    template<typename It, typename End>
      constexpr explicit(extent != dynamic_extent)
      span(It first, End last);
    ...
  };
}
```

As a consequence, implicit type conversions of spans are allowed only if the spans have a dynamic extent.

22.4 Afternotes

Making `typename` optional in certain situations was first proposed by by Daveed Vandevoorde in `http://wg21.link/p0634r0`. The finally accepted wording was formulated by Nina Ranns and Daveed Vandevoorde in `http://wg21.link/p0634r3`.

Class template argument deduction for aggregates was first proposed by Mike Spertus in `http://wg21.link/p1021r0`. The finally accepted wording was formulated by Timur Doumler in `http://wg21.link/p1816r0` and in `http://wg21.link/p2082r1`.

The conditional `explicit` feature was first proposed by Barry Revzin and Stephan T. Lavavej in `http://wg21.link/p0892r0`. The finally accepted wording was formulated by Barry Revzin and Stephan T. Lavavej in `http://wg21.link/p0892r2`.

This page is intentionally left blank

Chapter 23

Small Improvements for the C++ Standard Library

This chapter presents additional features and extensions of the C++20 standard library that have not been covered in this book yet.

23.1 Updates for String Types

In C++20, some aspects of string types changed. These changes affect strings (type `std::basic_string<>` with its instantiations, such as `std::string`), string views (`std::basic_string_view<>` with its instantiations, such as `std::string_view`), or both.

In fact, C++20 introduces the following improvements for string types:

- All string types now support the spaceship operator `<=>`. For this, they now declare only `operator==` and `operator<=>` and no longer declare `operator!=`, `operator<`, `operator<=`, `operator>`, and `operator>=`.
- All string types now provide the new member functions `starts_with()` and `ends_with()`.
- For strings, the member function `reserve()` can no longer be used to request to shrink the capacity (memory allocated for the value) of strings. For this reason, you can no longer pass no argument to `reserve()`.
- For UTF-8 characters, C++ now provides the string types `std::u8string` and `std::u8string_view`. They are defined as `std::basic_string<>` and `std::basic_string_view<>` for the new UTF-8 character type `char8_t`. For this reason, library functions that return a UTF-8 string now have the return type `std::u8string`. Note that this change might break existing code when switching to C++20.
- Strings (`std::string` and other instantiations of `std::basic_string<>`) are `constexpr` now, meaning that you can use strings at compile time.

  Note that you cannot use a `std::string` at both compile time and runtime. However, there are ways to export a compile-time string to the runtime.
- String views are now marked as views and borrowed ranges.

- Standard hash functions were added for types `std::u8string` and `std::u8string_view`, as well as for `std::pmr::string`, `std::pmr::u8string`, `std::pmr::u16string`, `std::pmr::u32string`, and `std::pmr::wstring`.

The following sections explain the non-trivial improvements that are not introduced and explained in other chapters.

23.1.1 String Members `starts_with()` and `ends_with()`

Both strings and string views now have new member functions `starts_with()` and `ends_with()`. They provide an easy way to check the leading and trailing character of a string against a certain sequence of characters. You can compare against a single character, an array of characters, or a string or string view.

For example:

```
void foo(const std::string& s, std::string_view suffix)
{
  if (s.starts_with('.')) {
    ...
  }
  if (s.ends_with(".tmp")) {
    ...
  }
  if (s.ends_with(suffix)) {
    ...
  }
}
```

23.1.2 Restricted String Member `reserve()`

For strings, the member function `reserve()` can no longer be used to request to shrink the capacity (memory allocated for the value) of strings:

```
void modifyString(std::string& s)
{
  if (...) {
    s.clear();
    s.reserve(0);        // may no longer shrink memory (as before on some platforms)
    return;
  }
  ...
}
```

The reason for this is that releasing memory might take some time, meaning that the performance of this call could vary significantly when porting code.

For this reason, passing no argument to `reserve()` is no longer supported:

```
    s.reserve();         // ERROR since C++20
```

Use `shrink_to_fit()` instead:

```
    s.shrink_to_fit();   // still OK
```

23.2 std::source_location

Sometimes, it is important to let the program deal with the location of the source code that is currently being processed. This is especially used for logging, testing, and checking invariants. So far, programmers have had to use the C preprocessor macros `__FILE__`, `__LINE__`, and `__func__`. C++20 introduces a type-safe feature for that so that an object can be initialized with the current source location and this information can be passed around just like any other object.

The use is simple:

```
#include <source_location>

void foo()
{
  auto sl = std::source_location::current();
  ...
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}
```

The static `consteval` function `std::source_location::current()` yields an object for the current source location of type `std::source_location` with the following interface:

- `file_name()` yields the name of the file.
- `function_name()` yields the name of the function (empty if called outside any function).
- `line()` yields the line number (may be `0` when the line number is not known).
- `column()` yields the column in the line (may be `0` when the column number is not known).

Details such as the exact format of the function name and the exact position of the column might differ. For example, with the GCCs library, the output might look as follows:

```
file:     sourceloc.cpp
function: void foo()
line/col: 8/42
```

while the output of Visual C++ might look as follows:

```
file:     sourceloc.cpp
function: foo
line/col: 8/35
```

Note that by using `std::source_location::current()` as a default argument in a parameter declaration, you get the location of the function call. For example:

```
void bar(std::source_location sl = std::source_location::current())
{
  ...
  std::cout << "file:     " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}
```

```
int main()
{
  ...
  bar();
  ...
}
```

The output might be something like:

```
file:     sourceloc.cpp
function: int main()
line/col: 34/6
```

or:

```
file:     sourceloc.cpp
function: main
line/col: 34/3
```

Because a `std::source_location` is an object, you can store it in a container and pass it around:

```
std::source_location myfunc()
{
  auto sl = std::source_location::current();
  ...
  return sl;
}

int main()
{
  std::vector<std::source_location> locs;
  ...
  locs.push_back(myfunc());
  ...
  for (const auto& loc : locs) {
    std::cout << "called: " << loc.function_name() << '\n';
  }
}
```

The output might be:

```
called: 'std::source_location myfunc()'
```

or:

```
called: 'myfunc'
```

See *lib/sourceloc.cpp* for the complete example.

23.3 Safe Comparisons of Integral Values and Sizes

Comparing integral values of different types is more complex than expected. This section describes two new features of C++20 for dealing with this problem:

- Utilities for integral comparisons
- `std::ssize()`

23.3.1 Safe Comparisons of Integral Values

Comparing and converting numbers, even of different numeric types, should be a trivial task. Unfortunately, it is not. Almost all programmers have seen warnings about code doing this. These warnings have a good reason: because of implicit conversions, we may write unsafe code without noticing it.

For example, most of the time we expect that a simple `x < y` just works. Consider the following example:[1]

```
int x = -7;
unsigned y = 42;

if (x < y) ...                       // OOPS: false
```

The reason for this behavior is that when comparing a signed with an unsigned value by rule (from the programming language C), the signed value is converted into an unsigned type, which makes it a big positive integral value.

To fix this code, you have to write the following instead:

```
if (x < static_cast<int>(y)) ...   // true
```

Other problems can occur if we compare small integral values with large integral values.

C++20 now provides functions for safe integral comparisons in `<utility>`. Table *Safe integral comparison functions* lists these functions. You can use them to simply write:

```
if (std::cmp_less(x, y)) ...       // true
```

This integral comparison is always safe.

Constant	**Template**
`std::`**`cmp_equal`**`(`*x*`,` *y*`)`	Yields whether *x* is equal to *y*
`std::`**`cmp_not_equal`**`(`*x*`,` *y*`)`	Yields whether *x* is not equal to *y*
`std::`**`cmp_less`**`(`*x*`,` *y*`)`	Yields whether *x* is less than *y*
`std::`**`cmp_less_equal`**`(`*x*`,` *y*`)`	Yields whether *x* is less than or equal to *y*
`std::`**`cmp_greater`**`(`*x*`,` *y*`)`	Yields whether *x* is greater than *y*
`std::`**`cmp_greater_equal`**`(`*x*`,` *y*`)`	Yields whether *x* is greater than or equal to *y*
`std::`**`in_range<`***T***`>`**`(`*x*`)`	Yields whether *x* is a valid value for type *T*

Table 23.1. Safe integral comparison functions

[1] Thanks to Federico Kircheis for pointing out this example in `http://wg21.link/p0586r2`.

The function `std::in_range()` yields `true` if a passed value is a value that can be represented by the passed type:

```
bool b = std::in_range<int>(x);   // true if x has a valid int value
```

It simply yields whether `x` is greater than or equal to the minimum value and less than or equal to the maximum value of the passed type.

Note that these functions cannot be used to compare values of `bool`, character types, or `std::byte`.

23.3.2 `ssize()`

We often need the size of a collection, array, or range as a signed value. For example, to avoid a warning here:

```
for (int i = 0; i < coll.size(); ++coll) {         // possible warning
  ...
}
```

The problem is that `size()` yields an unsigned value and comparisons between signed and unsigned values might fail if the values are very high.

While you can declare `i` as an unsigned int or `std::size_t`, there might be a good reason to use it as `int`.

A helper function `std::ssize()` was introduced that allows the following use instead:

```
for (int i = 0; i < std::ssize(coll); ++coll) {    // usually no warning
  ...
}
```

Thanks to ADL, it is enough to write the following when a standard container or other standard types are used:

```
for (int i = 0; i < ssize(coll); ++coll) {         // OK for std types
  ...
}
```

Note that this is sometimes even necessary to avoid compile-time errors. An example is the initialization of latches, barriers, and semaphores.

Note also that the ranges library provides `ssize()` in the namespace `std::ranges`.

23.4 Mathematical Constants

C++20 introduces constants for the most important mathematical floating-point constants. Table *Math constants* lists them.

The constants are provided in the header file **`<numbers>`** in the namespace `std::numbers`.

Constant	Template
`std::numbers::e`	`std::numbers::e_v<>`
`std::numbers::pi`	`std::numbers::pi_v<>`
`std::numbers::inv_pi`	`std::numbers::inv_pi_v<>`
`std::numbers::inv_sqrtpi`	`std::numbers::inv_sqrtpi_v<>`
`std::numbers::sqrt2`	`std::numbers::sqrt2_v<>`
`std::numbers::sqrt3`	`std::numbers::sqrt3_v<>`
`std::numbers::inv_sqrt3`	`std::numbers::inv_sqrt3_v<>`
`std::numbers::log2e`	`std::numbers::log2e_v<>`
`std::numbers::log10e`	`std::numbers::log10e_v<>`
`std::numbers::ln2`	`std::numbers::ln2_v<>`
`std::numbers::ln10`	`std::numbers::ln10_v<>`
`std::numbers::egamma`	`std::numbers::egamma_v<>`
`std::numbers::phi`	`std::numbers::phi_v<>`

Table 23.2. Math constants

The constants are specializations for type `double` of corresponding variable templates that have the suffix `_v`. The values are the nearest representable values of the corresponding type. For example:

```
namespace std::number {
  template<std::floating_point T> inline constexpr T pi_v<T> = ... ;
  inline constexpr double pi = pi_v<double>;
}
```

As you can see, the definitions use the `std::floating_point` concept (which was introduced for that reason).

Therefore, you can use them as follows:

```
#include <numbers>
...

double area1 = rad * rad * std::numbers::pi;

long double area2 = rad * rad * std::numbers::pi_v<long double>;
```

23.5 Utilities for Dealing with Bits

C++20 provides better and cleaner support for dealing with bits:

- Missing low-level bit operations
- Bit casts
- Checks for the endianness of a platform

All of these utilities are defined in the header file `<bit>`.

23.5.1 Bit Operations

Hardware usually has special support for bit operations such as “rotate left” or “rotate right.” However, before C++20, C++ programmers did not have direct access to these instructions. The newly introduced bit operations provide a direct API to the bit instructions of the underlying CPU.

Table *Bit operations* lists all standardized *bit operations* that C++20 introduced. They are provided in the header file `<bit>` as free-standing functions in the namespace `std`.

Operation	Meaning
`rotl(`*val*, *n*`)`	Yields *val* with *n* bits rotated to the left
`rotr(`*val*, *n*`)`	Yields *val* with *n* bits rotated to the right
`countl_zero(`*val*`)`	Yields number of leading (most significant) `0` bits
`countl_one(`*val*`)`	Yields number of leading (most significant) `1` bits
`countr_zero(`*val*`)`	Yields number of trailing (least significant) `0` bits
`countr_one(`*val*`)`	Yields number of trailing (least significant) `1` bits
`popcount(`*val*`)`	Yields number of `1` bits in the value
`has_single_bit(`*val*`)`	Yields whether *val* is a power of `2` (one bit set)
`bit_floor(`*val*`)`	Yields previous power-of-two value
`bit_ceil(`*val*`)`	Yields next power-of-two value
`bit_width(`*val*`)`	Yields number of bits necessary to store the value

Table 23.3. Bit operations

Consider the following program:

lib/bitops8.cpp

```
#include <iostream>
#include <format>
#include <bitset>
#include <bit>

int main()
{
  std::uint8_t i8 = 0b0000'1101;
  std::cout
    << std::format("{0:08b} {0:3}\n", i8)                  // 00001101
    << std::format("{0:08b} {0:3}\n", std::rotl(i8, 2))    // 00110100
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, 1))    // 10000110
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, -1))   // 00011010
    << std::format("{}\n", std::countl_zero(i8))           // four leading zeros
    << std::format("{}\n", std::countr_one(i8))            // one trailing one
    << std::format("{}\n", std::popcount(i8))              // three ones
    << std::format("{}\n", std::has_single_bit(i8))        // false
    << std::format("{0:08b} {0:3}\n", std::bit_floor(i8))  // 00001000
    << std::format("{0:08b} {0:3}\n", std::bit_ceil(i8))   // 00010000
    << std::format("{}\n", std::bit_width(i8));            // 4
}
```

The program has the following output:

```
00001101  13
00110100  52
10000110 134
00011010  26
4
1
3
false
00001000   8
00010000  16
4
```

Note the following:

- All these functions are provided only if the passed type is an ***unsigned*** integral type.
- The rotate functions also take a negative *n*, which means that the rotation changes its direction.
- All functions that return a count have the return type `int`. The only exception is `bit_width()`, which returns a value of the passed type, which is an inconsistency (I would assume a bug) in the standard. Therefore, when using it as an `int` or printing it directly, you might have to use a static cast.

If you run a corresponding program for a `std::uint16_t` (see *lib/bitops16.cpp*), you get the following output:

```
0000000000001101    13
0000000000110100    52
1000000000000110 32774
0000000000011010    26
12
1
3
false
0000000000001000     8
0000000000010000    16
4
```

Note also that these functions are defined only for *unsigned integral types*. That means:[2]

- You cannot use the bit operations for signed integral types:

  ```
  int b1 =  ... ;
  auto b2 = std::rotl(b1, 2);    // ERROR
  ```

- You cannot use the bit operations for type `char`:

  ```
  char b1 =  ... ;
  auto b2 = std::rotl(b1, 2);    // ERROR
  ```

  Type `unsigned char` works fine.

[2] Thanks to JeanHeyd Meneide for pointing this out.

- You cannot use the bit operations for type `std::byte`:

```
std::byte b1{...};
auto b2 = std::rotl(b1, 2);    // ERROR
```

Table *Hardware support for bit operations*, taken from `http://wg21.link/p0553r4`, lists the possible mapping of some of the new bit operations to existing hardware.

Operation	**Intel/AMD**	**ARM**	**PowerPC**
`rotl()`	ROL	–	`rldicl`
`rotr()`	ROR	ROR, EXTR	–
`popcount()`	POPCNT	–	`popcntb`
`countl_zero()`	BSR, LZCNT	CLZ	`cntlzd`
`countl_one()`	–	CLS	–
`countr_zero()`	BSF, TZCNT	–	–
`countr_one()`	–	–	–

Table 23.4. Hardware support for bit operations

23.5.2 `std::bit_cast<>()`

C++20 provides a new cast operation for changing the type of a sequence of bits. In contrast to using `reinterpret_cast<>` or unions, the operator `std::bit_cast<>` ensures that the number of bits fits, a standard layout is used, and no pointer type is used.

For example:

```
std::uint8_t b8 = 0b0000'1101;

auto bc = std::bit_cast<char>(b8);           // OK
auto by = std::bit_cast<std::byte>(b8);      // OK
auto bi = std::bit_cast<int>(b8);            // ERROR: wrong number of bits
```

23.5.3 `std::endian`

C++20 introduces a new utility enumeration type `std::endian`, which can be used to check the endianness of the execution environment. It introduces three enumeration values:

- `std::endian::big`, a value that stands for "big-endian" (scalar types are stored with the most significant byte placed first and the rest in descending order)
- `std::endian::little`, a value that stands for "little-endian" (scalar types are stored with the least significant byte placed first and the rest in ascending order)
- `std::endian::native`, a value that specifies the endianness of the execution environment

If all scalar types are big-endian, `std::endian::native` is equal to `std::endian::big`. If all scalar types are little-endian, `std::endian::native` is equal to `std::endian::little`. Otherwise, `std::endian::native` has a value that is neither `std::endian::big` nor `std::endian::little`.

If all scalar types have a size of 1, then `std::endian::little`, `std::endian::big`, and `std::endian::native` all have the same value.

The type is defined in the header file `<bit>`.

As enumeration values, these values can be used at compile time. For example:

```
#include <bit>
...

if constexpr (std::endian::native == std::endian::big) {
   ...        // handle big-endian
}
else if constexpr (std::endian::native == std::endian::little) {
   ...        // handle little-endian
}
else {
   ...        // handle mixed endian
}
```

23.6 `<version>`

C++20 introduces a new header file **`<version>`**. This header file provides no active functionality. Instead, it provides all implementation-specific general information about the C++ standard library being used.

For example, `<version>` might contain:

- The version and release date of the C++ standard library
- Copyright information

In addition, `<version>` defines the feature test macros of the C++ standard library (they are also defined in the header files they apply to).

Because this header file is short and fast to load, tools can include this header file to get all necessary information to make decisions based on the feature set provided or find all information they need to deal with general information of the library used.

23.7 Extensions for Algorithms

For algorithms, C++20 provides a couple of extensions (some of which have already been described in other chapters of this book).

23.7.1 Range Support

As discussed in the chapter about ranges, for many algorithms, there is now support:

- For passing whole ranges (containers, views) as a single argument
- For passing a projection parameter as a single argument

This support requires the algorithms to be called in the namespace `std::ranges`.

For the following algorithms, there is no support for ranges (yet):

- Numeric algorithms (such as `accumulate()`)
- Algorithms with parallel execution
- Algorithm `lexicographical_compare_three_way()`

For more details of the range support for all standard algorithms, see the algorithm overview.

23.7.2 New Algorithms

Table *New standard algorithms* lists the standard algorithms that were introduced with C++20.

Name	Effect
`min()`	Yield the minimum value of a passed ranged
`max()`	Yield the maximum value of a passed ranged
`minmax()`	Yield the minimum and maximum value of a passed ranged
`shift_left()`	Shift all elements to the front
`shift_right()`	Shift all elements to the back
`lexicographical_compare_three_way()`	Sort two ranges by using operator <=>

Table 23.5. New standard algorithms

`min()`, `max()`, and `minmax()` for Ranges

The algorithms `std::ranges::min()`, `std::ranges::max()`, and `std::ranges::minmax()` were introduced with the ranges library to yield the minimum and/or maximum value of a passed range. You can optionally pass a comparison criterion and a projection.

For example:

lib/minmax.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector coll{0, 8, 15, 47, 11};

  std::cout << std::ranges::min(coll) << '\n';
  std::cout << std::ranges::max(coll) << '\n';
  auto [min, max] = std::ranges::minmax(coll);
  std::cout << min << ' ' << max << '\n';
}
```

The program has the following output:

```
0
47
0 47
```

Note that there are no corresponding algorithms in `std` that take two iterators. There are only functions that take two values or a `std::initializer_list<>` (and a comparison criterion) as arguments. As usual, the ranges library also provides them with the option to pass a projection.

`shift_left()` and `shift_right()`

With the new standard algorithms `shift_left()` and `shift_right`, you can move elements to the front or back, respectively. The algorithms return the new end or beginning, respectively.

For example:

lib/shift.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ranges>

void print (const auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
int main()
{
  std::vector coll{1, 2, 3, 4, 5, 6, 7, 8};

  print(coll);

  // shift one element to the front (returns new end):
  std::shift_left(coll.begin(), coll.end(), 1);
  print(coll);

  // shift three elements to the back (returns new begin):
  auto newbeg = std::shift_right(coll.begin(), coll.end(), 3);
  print(coll);
  print(std::ranges::subrange{newbeg, coll.end()});
}
```

The program has the following output:

```
1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 8
2 3 4 2 3 4 5 6
2 3 4 5 6
```

You can pass an execution policy to allow the use of multiple threads. However, the support for ranges with passing a single-argument range and/or projects is not provided (probably an oversight).

`lexicographical_compare_three_way()`

To compare the elements of two different containers in such a way that one of the new comparison category types is returned, the algorithm `std::lexicographical_compare_three_way()` was introduced with C++20. It is described in the chapter about the operator <=>.

Note that for this algorithm, there is no support for ranges (yet). You can neither pass a range as a single argument nor pass projection parameters (this is probably an oversight).

23.7.3 `unseq` Execution Policy for Algorithms

C++17 introduced various execution policies for the newly introduced parallel algorithms. You could enable parallel computing and allow threads to operate on multiple data items in parallel (called *vectorization* or *SIMD processing*).

However, you could not allow algorithms to operate on multiple data items but restrict the algorithms to use only one thread. For this, C++20 now provides the execution policy `std::execution::unseq`. As usual, the new execution policy is a `constexpr` object of a corresponding unique class `unsequenced_policy` in the namespace `std::execution`. Table *Execution policies* lists all standardized *execution policies* now supported.

Policy	**Meaning**
`std::execution::seq`	Sequential execution of a single value with one thread
`std::execution::par`	Parallel execution of a single value with multiple threads
`std::execution::unseq`	Parallel execution of multiple values with one thread (since C++20)
`std::execution::par_unseq`	Parallel execution of multiple values with multiple threads

Table 23.6. Execution policies

The unsequenced execution policy allows the execution of the operations passed to access the elements to be interleaved on a single thread of execution. You should not use this policy with blocking synchronization (such as using mutexes) because that might result in deadlocks.

Here is an example that uses it:

lib/unseq.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <execution>

int main (int argc, char** argv)
{
  // init number of argument from command line (default: 1000):
  int numElems = 1000;
  if (argc > 1) {
    numElems = std::atoi(argv[1]);
  }

  // init vector of different double values:
  std::vector<double> coll;
  coll.reserve(numElems);
  for (int i=0; i<numElems; ++i) {
    coll.push_back(i * 4.37);
  }

  // process square roots:
  // - allow SIMD processing but only one thread
  std::for_each(std::execution::unseq,      // since C++20
                coll.begin(), coll.end(),
                [](auto& val) {
                  val = std::sqrt(val);
                });
  for (double value : coll) {
    std::cout << value << '\n';
  }
}
```

As usual for execution policies, you have no way of impacting when and how the policy is used. With this policy, you ***enable*** vectorization or SIMD computing; however, you do not mandate it. On hardware that does not support this policy, or if the implementation decides at runtime not to use it (e.g., because the CPU load is too high), the `unseq` policy results in sequential execution.

23.8 Afternotes

The string member functions `starts_with()` and `ends_with()` were accepted as proposed by Mikhail Maltsev in http://wg21.link/p0457r2.

The restriction of the `reserve()` function of strings was first proposed by Andrew Luo in http://wg21.link/lwg2968. The finally accepted wording was formulated by Mark Zeren and Andrew Luo in http://wg21.link/p0966r1.

Access to the source location was first proposed by Robert Douglas in http://wg21.link/n3972. The finally accepted wording was formulated by Robert Douglas, Corentin Jabot, Daniel Krügler, and Peter Sommerlad in http://wg21.link/p1208r6.

The functions for safe integral comparisons were accepted as proposed by Federico Kircheis in http://wg21.link/p0586r2.

The `ssize()` functions were discussed for a long time as a fix to the problem of comparing signed indexes with unsigned sizes. We would like to let all `size()` functions yield a signed value; however, there is the backward compatibility problem. As a reaction to a `std::span` proposal to use a signed size type, `ssize()` was first proposed by Robert Douglas, Nevin Liber, and Marshall Clow in http://wg21.link/p1089r0. The finally accepted wording was formulated by Jorg Brown in http://wg21.link/p1227r2. `std::ranges::ssize()` was later added by Hannes Hauswedell, Jorg Brown, and Casey Carter in http://wg21.link/p1970r2.

The mathematical constants were first proposed by Lev Minkovsky in http://wg21.link/p0631r0. The finally accepted wording was formulated by Lev Minkovsky and John McFarlane in http://wg21.link/p0631r8.

Bit operations were first proposed by Matthew Fioravante in http://wg21.link/n3864. The finally accepted wording was formulated by Jens Maurer in http://wg21.link/p0553r4 and http://wg21.link/p0556r3. The names were later modified as proposed by Vincent Reverdy in http://wg21.link/p1956r1.

`std::bit_cast<>()` was introduced as finally proposed by JF Bastien in http://wg21.link/p0476r2.

`std::endian` was first proposed by Howard Hinnant in http://wg21.link/p0463r0. The finally accepted wording was formulated by Howard Hinnant in http://wg21.link/p0463r1. The header file was later corrected as proposed by Walter E. Brown and Arthur O'Dwyer in http://wg21.link/p1612r1.

The header file `<version>` was accepted as proposed by Alan Talbot in http://wg21.link/p0754r2.

The new algorithms `shift_left()` and `shift_right()` were accepted as proposed by Dan Raviv in http://wg21.link/p0769r2. The new min/max algorithms for ranges were introduced with the ranges library as proposed in http://wg21.link/p0896r4.

The `unseq` execution policy was first proposed by Arch D. Robison, Pablo Halpern, Robert Geva, and Clark Nelson in http://wg21.link/p0076r0. The finally accepted wording was formulated by Alisdair Meredith and Pablo Halpern in http://wg21.link/p1001r2.

This page is intentionally left blank

Chapter 24

Deprecated and Removed Features

There are a few features that have been deprecated in C++20 or finally removed.

Implementations might still provide removed features but you cannot rely on that. Implementations might or might not issue a warning if you use deprecated features.

24.1 Deprecated and Removed Core Language Features

- The implicit capture of `*this` is deprecated.
- Aggregates may no longer have any user-declared (but not user-provided) constructors.

24.2 Deprecated and Removed Library Features

24.2.1 Deprecated Library Features

The following library features are deprecated since C++20 and should no longer be used:

- The type trait `is_pod<>`.
  Use `is_trivial<>` or a similar type trait instead.
- Atomic operations for regular shared pointers are deprecated now. Use atomic shared pointers instead.

24.2.2 Removed Library Features

- For strings, the member function `reserve()` can no longer be called with no argument and it no longer shrinks the capacity.
- You can no longer write a UTF-8 string to a standard output stream.

24.3 Afternotes

Deprecating `is_pod<>` was accepted as proposed by Jens Maurer in `http://wg21.link/p0767r1`.

Glossary

This glossary provides a short definition of the most important non-trivial technical terms used in this book.

A

aggregate

Aggregates are simple `class` or `struct` types, with restrictions so that the focus lies on storing data and not providing complex behavior. Coming from the programming language C, they are historically raw arrays, `structs`, and combinations of these two features, which you can initialize with curly braces. In C++, the key requirement is that aggregates have only public data members, no constructors, and no virtual functions. Note that these requirements have changed slightly in C++20. Note also that you can now initialize aggregates with parentheses.

argument-dependent lookup (ADL)

A feature that allows the programmer to skip namespace qualification of a function when one of the parameters is in its namespace. This means that a function is also looked up in all namespaces of the arguments passed.

C

class template argument deduction (CTAD)

The process that implicitly determines template arguments from the context in which class templates are used. It was introduced with C++17 and allows you to skip the specification of template arguments of an object when the template parameters can be deduced from the constructor.

F

forwarding reference

The term that the C++ standard uses for *universal references*.

full specialization

An alternative definition for a (*primary*) template that no longer depends on any template parameter.

function object (functor)

An object that can be used as a function. For this, `operator()` is defined in the class. All lambdas are function objects.

G

glvalue

A *value category* of expressions that produce a location for a stored value (generalized localizable value). A glvalue can be an *lvalue* or an *xvalue*.

I

incomplete type

A class, struct, or unscoped enumeration type that is declared but not defined, an array of unknown size, `void` (optionally with `const` and/or `volatile`), or an array of an incomplete element type.

L

lvalue

A *value category* of expressions that produce a location for a stored value that is not assumed to be movable (i.e., *glvalues* that are not *xvalues*). Examples are:

- Named objects (variables)
- String literals
- Returned lvalue references
- Functions and all references to functions
- Data members of lvalues

P

partial specialization

An alternative definition for a (*primary*) template that still depends on one or more template parameters.

predicate

A callable (function, function object, or lambda) that checks whether a certain criterion applies to one or more arguments. It returns a Boolean value, is read-only, and *stateless*.

prvalue

A *value category* of expressions that perform initializations. Prvalues can be assumed to designate pure mathematical values such as `1` or `true` and temporary objects without names. Examples are:

- All literals except string literals (`42`, `true`, `nullptr`, etc.)
- Returned values (values not returned by reference)
- Results of constructor calls
- Lambdas
- `this`

What was called an *rvalue* before C++11 is called a *prvalue* since C++11.

R

resource acquisition is initialization (RAII)

A programming pattern to delegate clean-ups necessary for the end of using a resource to a destructor so that the clean-ups happen automatically when the object that represents the resource leaves its scope or ends its lifetime.

regular type

A type that matches the semantics of built-in value types (such as `int`). Based on the definition in `http://stepanovpapers.com/DeSt98.pdf`, a regular type provides the following basic operations:

- Default construction (`T x;`)
- Copying (`T y = x;`) and in C++, moving
- Assignment (`x = y;`) and in C++, move assignment
- Equality and inequality (`x == y` and `x != y`)
- Ordering (`x < y` etc.)

These operations follow the “usual” naive rules:

- If one object is a copy of the other, the objects are equal.
- A copy of an object is equal to an object created with the default constructor to which the source value was assigned.
- Objects have value semantics. If two objects are equal and we modify one of them, they are no longer equal.

If only the comparison operators are missing, the type is called *semiregular*.

rvalue

A *value category* of expressions that are not *lvalues*. An rvalue can be a *prvalue* (such as a temporary object without a name) or an *xvalue* (e.g., an *lvalue* marked with `std::move()`). What was called an *rvalue* before C++11 is called a *prvalue* since C++11.

S

semiregular type

A type that is *regular* but does not provide comparison operators:

- If one object is a copy of the other, the objects are equal.
- A copy of an object is equal to an object created with the default constructor to which the source value was assigned.
- Objects have value semantics. If two objects are equal and we modify one of them, they are no longer equal.

substitution failure is not an error (SFINAE)

A mechanism that silently discards templates instead of triggering compilation errors when arguments to template parameters make their declarations ill-formed. You can use the mechanism to disable templates for certain arguments by forcing ill-formed declarations then.

small string optimization (SSO)

An approach to save allocating memory for short strings by always reserving memory for a certain number of characters. A typical value in standard library implementations is to always reserve 16 or 24 bytes of memory so that the string can have 15 or 23 characters (plus 1 byte for the null terminator) without allocating memory. This makes all string objects larger but usually saves a lot of running time because in practice, strings are often shorter than 16 or 24 characters and allocating memory on the heap is quite an expensive operation.

stateless

A function or operation is *stateless* if it does not change its state due to a call. This has the effect that it does not change its behavior over time and always yields the same result for the same arguments.

standard template library (STL)

The STL is the part of the C++ standard library that deals with containers (data structures), algorithms, and as glue, the iterators. This approach was adopted for the first C++ standard with the goal that programmers can benefit from various standard data structures and algorithms without having to implement them. As generic code, you still get compile-time error messages when you try to combine things that are not supported.

U

universal reference

A reference that can universally refer to any object while not making it `const`. It can also extend the lifetime of return values. It is declared as an rvalue reference of a template parameter (`T&&`) or with `auto&&`.
Universal references are useful for perfectly forwarding arguments with `std::forward<>()` (for that reason, the C++ standard names them *forwarding references*).
However, they also are the only kind of references that can refer to any expression (both *lvalues* and *rvalues*) without making the value `const`. For this reason, they are necessary to declare parameters of generic functions that take views.

V

value category

A classification of expressions. The traditional value categories *lvalues* and *rvalues* were inherited from the programming language C. C++11 introduced additional categories, meaning that C++ now has the following primary value categories:

– *prvalues* (pure rvalues), which are values used to initialize objects (including parameters)
– *lvalues* (localizable values), which are objects you can ask for the address
– *xvalues* (eXpiring values), which are objects where we no longer need the value (usually, objects marked with `std::move()`)

In addition, C++ has the following combined value categories:

– *glvalues* (generalized lvalues), which means “either lvalue or xvalue”
– *rvalues* (readable values), which means “either rvalue or xvalue”

variable template

A generic variable. It allows us to define variables or static members by substituting the template parameters with specific types or values.

variadic template

A template with a template parameter that represents an arbitrary number of types or values.

X

xvalue

A *value category* of expressions that produce a location for a stored object that can be assumed to be no longer needed. Examples are:

– Values marked with `std::move()`
– Returned rvalue references
– Casts of objects (not of functions) to an rvalue reference
– Value members of rvalues

This page is intentionally left blank

Index